Finite difference methods for wave motion

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Oct 5, 2014

This is still a **preliminary version**.

Contents

1 Simulation of waves on a string **5**
1.1 Discretizing the domain . . . 6
1.2 The discrete solution . . . 6
1.3 Fulfilling the equation at the mesh points . . . 6
1.4 Replacing derivatives by finite differences . . . 7
1.5 Formulating a recursive algorithm . . . 8
1.6 Sketch of an implementation . . . 10

2 Verification **10**
2.1 A slightly generalized model problem . . . 11
2.2 Using an analytical solution of physical significance . . . 11
2.3 Manufactured solution . . . 12
2.4 Constructing an exact solution of the discrete equations . . . 14

3 Implementation **15**
3.1 Making a solver function . . . 16
3.2 Verification: exact quadratic solution . . . 17
3.3 Visualization: animating the solution . . . 17
3.4 Running a case . . . 20
3.5 The benefits of scaling . . . 21

4 Vectorization **22**
4.1 Operations on slices of arrays . . . 22
4.2 Finite difference schemes expressed as slices . . . 24
4.3 Verification . . . 25
4.4 Efficiency measurements . . . 26
4.5 Remark on the updating of arrays . . . 26

5 Exercises **27**

6 Generalization: reflecting boundaries **30**
6.1 Neumann boundary condition 30
6.2 Discretization of derivatives at the boundary 31
6.3 Implementation of Neumann conditions 32
6.4 Index set notation 33
6.5 Alternative implementation via ghost cells 35

7 Generalization: variable wave velocity **37**
7.1 The model PDE with a variable coefficient 37
7.2 Discretizing the variable coefficient 38
7.3 Computing the coefficient between mesh points 38
7.4 How a variable coefficient affects the stability 39
7.5 Neumann condition and a variable coefficient 40
7.6 Implementation of variable coefficients 41
7.7 A more general model PDE with variable coefficients 42
7.8 Generalization: damping 42

8 Building a general 1D wave equation solver **43**
8.1 User action function as a class 43
8.2 Pulse propagation in two media 44

9 Exercises **46**

10 Analysis of the difference equations **51**
10.1 Properties of the solution of the wave equation 51
10.2 More precise definition of Fourier representations 53
10.3 Stability 54
10.4 Numerical dispersion relation 57
10.5 Extending the analysis to 2D and 3D 59

11 Finite difference methods for 2D and 3D wave equations **63**
11.1 Multi-dimensional wave equations 63
11.2 Mesh 65
11.3 Discretization 65

12 Implementation **67**
12.1 Scalar computations 68
12.2 Vectorized computations 70
12.3 Verification 72

13 Migrating loops to Cython **73**
13.1 Declaring variables and annotating the code 73
13.2 Visual inspection of the C translation 75
13.3 Building the extension module 76
13.4 Calling the Cython function from Python 77

14 Migrating loops to Fortran **78**
14.1 The Fortran subroutine 78
14.2 Building the Fortran module with f2py 79
14.3 How to avoid array copying 81

15 Migrating loops to C via Cython **82**
15.1 Translating index pairs to single indices 83
15.2 The complete C code 83
15.3 The Cython interface file 84
15.4 Building the extension module 85

16 Migrating loops to C via f2py **86**
16.1 Migrating loops to C++ via f2py 87

17 Using classes to implement a simulator **87**

18 Exercises **87**

19 Applications of wave equations **89**
19.1 Waves on a string 89
19.2 Waves on a membrane 92
19.3 Elastic waves in a rod 92
19.4 The acoustic model for seismic waves 93
19.5 Sound waves in liquids and gases 94
19.6 Spherical waves 96
19.7 The linear shallow water equations 97
19.8 Waves in blood vessels 99
19.9 Electromagnetic waves 101

20 Exercises **101**

List of Exercises, Problems, and Projects

Exercise	1	Simulate a standing wave	p. 27
Exercise	2	Add storage of solution in a user action function ...	p. 28
Exercise	3	Use a class for the user action function	p. 28
Exercise	4	Compare several Courant numbers in one movie	p. 28
Project	5	Calculus with 1D mesh functions	p. 29
Exercise	6	Find the analytical solution to a damped wave ...	p. 46
Problem	7	Explore symmetry boundary conditions	p. 47
Exercise	8	Send pulse waves through a layered medium	p. 47
Exercise	9	Compare discretizations of a Neumann condition ...	
Exercise	10	Verification by a cubic polynomial in space	p. 48
Exercise	11	Check that a solution fulfills the discrete ...	p. 87
Project	12	Calculus with 2D/3D mesh functions	p. 88
Exercise	13	Implement Neumann conditions in 2D	p. 88
Exercise	14	Test the efficiency of compiled loops in 3D	p. 89
Exercise	15	Simulate waves on a non-homogeneous string	p. 101
Exercise	16	Simulate damped waves on a string	p. 102
Exercise	17	Simulate elastic waves in a rod	p. 102
Exercise	18	Simulate spherical waves	p. 102
Exercise	19	Explain why numerical noise occurs	p. 103
Exercise	20	Investigate harmonic averaging in a 1D model	p. 103
Problem	21	Implement open boundary conditions	p. 103
Exercise	22	Implement periodic boundary conditions	p. 105
Problem	23	Earthquake-generated tsunami over a subsea ...	p. 106
Problem	24	Earthquake-generated tsunami over a 3D hill	p. 108
Problem	25	Investigate Matplotlib for visualization	p. 109
Problem	26	Investigate visualization packages	p. 109
Problem	27	Implement loops in compiled languages	p. 109
Exercise	28	Simulate seismic waves in 2D	p. 110
Project	29	Model 3D acoustic waves in a room	p. 110
Project	30	Solve a 1D transport equation	p. 111
Problem	31	General analytical solution of a 1D damped ...	p. 114
Problem	32	General analytical solution of a 2D damped ...	p. 116

A very wide range of physical processes lead to wave motion, where signals are propagated through a medium in space and time, normally with little or no permanent movement of the medium itself. The shape of the signals may undergo changes as they travel through matter, but usually not so much that the signals cannot be recognized at some later point in space and time. Many types of wave motion can be described by the equation $u_{tt} = \nabla \cdot (c^2 \nabla u) + f$, which we will solve in the forthcoming text by finite difference methods.

1 Simulation of waves on a string

We begin our study of wave equations by simulating one-dimensional waves on a string, say on a guitar or violin string. Let the string in the deformed state coincide with the interval $[0, L]$ on the x axis, and let $u(x, t)$ be the displacement at time t in the y direction of a point initially at x. The displacement function u is governed by the mathematical model

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \qquad x \in (0, L),\ t \in (0, T] \tag{1}$$

$$u(x, 0) = I(x), \qquad x \in [0, L] \tag{2}$$

$$\frac{\partial}{\partial t} u(x, 0) = 0, \qquad x \in [0, L] \tag{3}$$

$$u(0, t) = 0, \qquad t \in (0, T] \tag{4}$$

$$u(L, t) = 0, \qquad t \in (0, T] \tag{5}$$

The constant c and the function $I(x)$ must be prescribed.

Equation (1) is known as the one-dimensional *wave equation*. Since this PDE contains a second-order derivative in time, we need *two initial conditions*, here (2) specifying the initial shape of the string, $I(x)$, and (3) reflecting that the initial velocity of the string is zero. In addition, PDEs need *boundary conditions*, here (4) and (5), specifying that the string is fixed at the ends, i.e., that the displacement u is zero.

The solution $u(x, t)$ varies in space and time and describes waves that are moving with velocity c to the left and right.

Sometimes we will use a more compact notation for the partial derivatives to save space:

$$u_t = \frac{\partial u}{\partial t}, \quad u_{tt} = \frac{\partial^2 u}{\partial t^2}, \tag{6}$$

and similar expressions for derivatives with respect to other variables. Then the wave equation can be written compactly as $u_{tt} = c^2 u_{xx}$.

The PDE problem (1)-(5) will now be discretized in space and time by a finite difference method.

1.1 Discretizing the domain

The temporal domain $[0, T]$ is represented by a finite number of mesh points

$$0 = t_0 < t_1 < t_2 < \cdots < t_{N_t-1} < t_{N_t} = T\,. \tag{7}$$

Similarly, the spatial domain $[0, L]$ is replaced by a set of mesh points

$$0 = x_0 < x_1 < x_2 < \cdots < x_{N_x-1} < x_{N_x} = L\,. \tag{8}$$

One may view the mesh as two-dimensional in the x, t plane, consisting of points (x_i, t_n), with $i = 0, \ldots, N_x$ and $n = 0, \ldots, N_t$.

Uniform meshes. For uniformly distributed mesh points we can introduce the constant mesh spacings Δt and Δx. We have that

$$x_i = i\Delta x,\ i = 0, \ldots, N_x, \quad t_i = n\Delta t,\ n = 0, \ldots, N_t\,. \tag{9}$$

We also have that $\Delta x = x_i - x_{i-1}$, $i = 1, \ldots, N_x$, and $\Delta t = t_n - t_{n-1}$, $n = 1, \ldots, N_t$. Figure 1 displays a mesh in the x, t plane with $N_t = 5$, $N_x = 5$, and constant mesh spacings.

1.2 The discrete solution

The solution $u(x, t)$ is sought at the mesh points. We introduce the mesh function u_i^n, which approximates the exact solution at the mesh point (x_i, t_n) for $i = 0, \ldots, N_x$ and $n = 0, \ldots, N_t$. Using the finite difference method, we shall develop algebraic equations for computing the mesh function. The circles in Figure 1 illustrate neighboring mesh points where values of u_i^n are connected through an algebraic equation. In this particular case, u_2^1, u_1^2, u_2^2, u_3^2, and u_2^3 are connected in an algebraic equation associated with the center point $(2, 2)$. The term *stencil* is often used about the algebraic equation at a mesh point, and the geometry of a typical stencil is illustrated in Figure 1. One also often refers to the algebraic equations as *discrete equations*, *(finite) difference equations* or a *finite difference scheme*.

1.3 Fulfilling the equation at the mesh points

For a numerical solution by the finite difference method, we relax the condition that (1) holds at all points in the space-time domain $(0, L) \times (0, T]$ to the requirement that the PDE is fulfilled at the *interior* mesh points:

$$\frac{\partial^2}{\partial t^2} u(x_i, t_n) = c^2 \frac{\partial^2}{\partial x^2} u(x_i, t_n), \tag{10}$$

for $i = 1, \ldots, N_x - 1$ and $n = 1, \ldots, N_t - 1$. For $n = 0$ we have the initial conditions $u = I(x)$ and $u_t = 0$, and at the boundaries $i = 0, N_x$ we have the boundary condition $u = 0$.

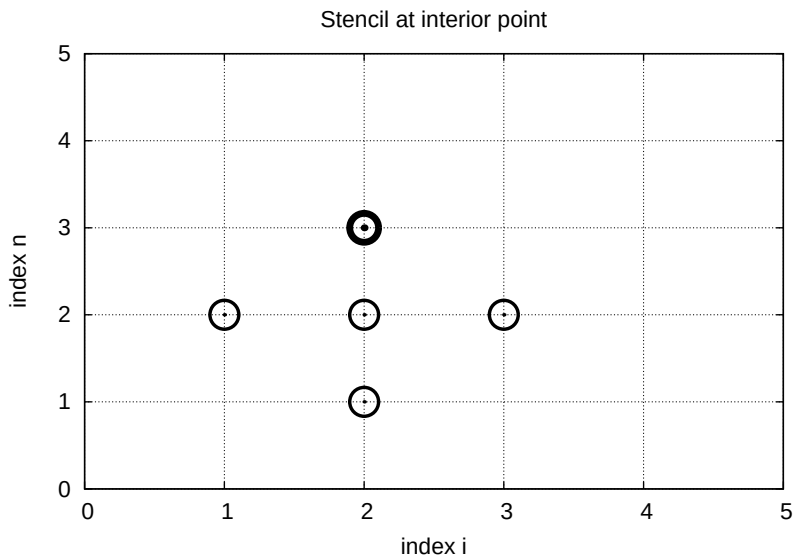

Figure 1: Mesh in space and time for a 1D wave equation.

1.4 Replacing derivatives by finite differences

The second-order derivatives can be replaced by central differences. The most widely used difference approximation of the second-order derivative is

$$\frac{\partial^2}{\partial t^2}u(x_i,t_n) \approx \frac{u_i^{n+1}-2u_i^n+u_i^{n-1}}{\Delta t^2}\,.$$

It is convenient to introduce the finite difference operator notation

$$[D_tD_tu]_i^n = \frac{u_i^{n+1}-2u_i^n+u_i^{n-1}}{\Delta t^2}\,.$$

A similar approximation of the second-order derivative in the x direction reads

$$\frac{\partial^2}{\partial x^2}u(x_i,t_n) \approx \frac{u_{i+1}^n-2u_i^n+u_{i-1}^n}{\Delta x^2} = [D_xD_xu]_i^n\,.$$

Algebraic version of the PDE. We can now replace the derivatives in (10) and get

$$\frac{u_i^{n+1}-2u_i^n+u_i^{n-1}}{\Delta t^2} = c^2\frac{u_{i+1}^n-2u_i^n+u_{i-1}^n}{\Delta x^2}, \tag{11}$$

or written more compactly using the operator notation:

$$[D_tD_tu = c^2D_xD_x]_i^n\,. \tag{12}$$

Algebraic version of the initial conditions. We also need to replace the derivative in the initial condition (3) by a finite difference approximation. A centered difference of the type

$$\frac{\partial}{\partial t}u(x_i,t_n) \approx \frac{u_i^1 - u_i^{-1}}{2\Delta t} = [D_{2t}u]_i^0,$$

seems appropriate. In operator notation the initial condition is written as

$$[D_{2t}u]_i^n = 0, \quad n = 0\,.$$

Writing out this equation and ordering the terms give

$$u_i^{n-1} = u_i^{n+1}, \quad i = 0,\ldots,N_x,\ n = 0\,. \tag{13}$$

The other initial condition can be computed by

$$u_i^0 = I(x_i), \quad i = 0,\ldots,N_x\,.$$

1.5 Formulating a recursive algorithm

We assume that u_i^n and u_i^{n-1} are already computed for $i = 0,\ldots,N_x$. The only unknown quantity in (11) is therefore u_i^{n+1}, which we can solve for:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right), \tag{14}$$

where we have introduced the parameter

$$C = c\frac{\Delta t}{\Delta x}, \tag{15}$$

known as the *Courant number*.

C is the key parameter in the discrete diffusion equation.

We see that the discrete version of the PDE features only one parameter, C, which is therefore the key parameter that governs the quality of the numerical solution (see Section 10 for details). Both the primary physical parameter c and the numerical parameters Δx and Δt are lumped together in C. Note that C is a dimensionless parameter.

Given that u_i^{n-1} and u_i^n are computed for $i = 0,\ldots,N_x$, we find new values at the next time level by applying the formula (14) for $i = 1,\ldots,N_x-1$. Figure 1 illustrates the points that are used to compute u_2^3. For the boundary points, $i = 0$ and $i = N_x$, we apply the boundary conditions $u_i^{n+1} = 0$.

A problem with (14) arises when $n = 0$ since the formula for u_i^1 involves u_i^{-1}, which is an undefined quantity outside the time mesh (and the time domain).

However, we can use the initial condition (13) in combination with (14) when $n = 0$ to arrive at a special formula for u_i^1:

$$u_i^1 = u_i^0 - \frac{1}{2}C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) . \tag{16}$$

Figure 2 illustrates how (16) connects four instead of five points: u_2^1, u_1^0, u_2^0, and u_3^0.

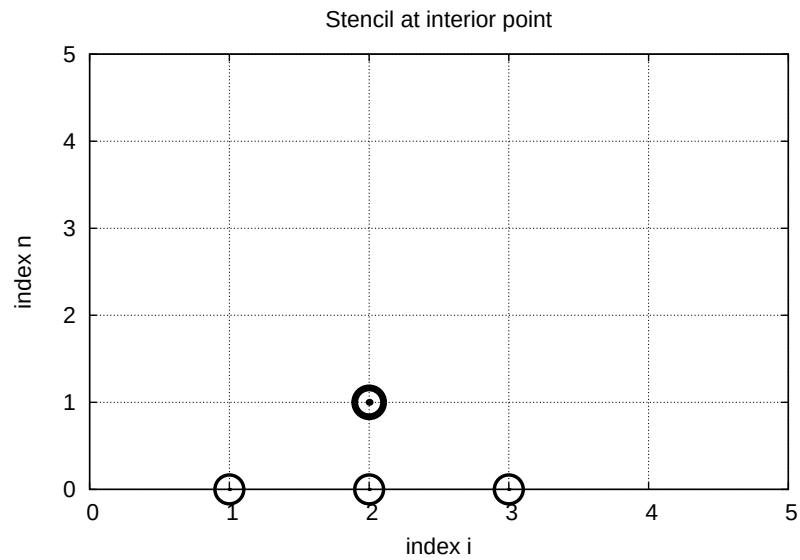

Figure 2: Modified stencil for the first time step.

We can now summarize the computational algorithm:

1. Compute $u_i^0 = I(x_i)$ for $i = 0, \ldots, N_x$
2. Compute u_i^1 by (16) and set $u_i^1 = 0$ for the boundary points $i = 0$ and $i = N_x$, for $n = 1, 2, \ldots, N - 1$,
3. For each time level $n = 1, 2, \ldots, N_t - 1$
   (a) apply (14) to find u_i^{n+1} for $i = 1, \ldots, N_x - 1$
   (b) set $u_i^{n+1} = 0$ for the boundary points $i = 0$, $i = N_x$.

The algorithm essentially consists of moving a finite difference stencil through all the mesh points, which is illustrated by an animation in a web page or a movie file.

1.6 Sketch of an implementation

In a Python implementation of this algorithm, we use the array elements `u[i]` to store u_i^{n+1}, `u_1[i]` to store u_i^n, and `u_2[i]` to store u_i^{n-1}. Our naming convention is use `u` for the unknown new spatial field to be computed, `u_1` as the solution at one time step back in time, `u_2` as the solution two time steps back in time and so forth.

The algorithm only needs to access the three most recent time levels, so we need only three arrays for u_i^{n+1}, u_i^n, and u_i^{n-1}, $i = 0, \ldots, N_x$. Storing all the solutions in a two-dimensional array of size $(N_x+1)\times(N_t+1)$ would be possible in this simple one-dimensional PDE problem, but is normally out of the question in three-dimensional (3D) and large two-dimensional (2D) problems. We shall therefore in all our programs for solving PDEs have the unknown in memory at as few time levels as possible.

The following Python snippet realizes the steps in the computational algorithm.

```
# Given mesh points as arrays x and t (x[i], t[n])
dx = x[1] - x[0]
dt = t[1] - t[0]
C = c*dt/dx            # Courant number
Nt = len(t)-1
C2 = C**2              # Help variable in the scheme

# Set initial condition u(x,0) = I(x)
for i in range(0, Nx+1):
    u_1[i] = I(x[i])

# Apply special formula for first step, incorporating du/dt=0
for i in range(1, Nx):
    u[i] = u_1[i] - 0.5*C**2(u_1[i+1] - 2*u_1[i] + u_1[i-1])
u[0] = 0;  u[Nx] = 0   # Enforce boundary conditions

# Switch variables before next step
u_2[:], u_1[:] = u_1, u

for n in range(1, Nt):
    # Update all inner mesh points at time t[n+1]
    for i in range(1, Nx):
        u[i] = 2u_1[i] - u_2[i] - \
               C**2(u_1[i+1] - 2*u_1[i] + u_1[i-1])

    # Insert boundary conditions
    u[0] = 0;  u[Nx] = 0

    # Switch variables before next step
    u_2[:], u_1[:] = u_1, u
```

2 Verification

Before implementing the algorithm, it is convenient to add a source term to the PDE (1) since it gives us more freedom in finding test problems for verification.

In particular, the source term allows us to use *manufactured solutions* for software testing, where we simply choose some function as solution, fit the corresponding source term, and define boundary and initial conditions consistent with the chosen solution. Such solutions will seldom fulfill the initial condition (3) so we need to generalize this condition to $u_t = V(x)$.

2.1 A slightly generalized model problem

We now address the following extended initial-boundary value problem for one-dimensional wave phenomena:

$$
\begin{aligned}
u_{tt} &= c^2 u_{xx} + f(x,t), && x \in (0,L),\ t \in (0,T] && (17)\\
u(x,0) &= I(x), && x \in [0,L] && (18)\\
u_t(x,0) &= V(x), && x \in [0,L] && (19)\\
u(0,t) &= 0, && t > 0 && (20)\\
u(L,t) &= 0, && t > 0 && (21)
\end{aligned}
$$

Sampling the PDE at (x_i, t_n) and using the same finite difference approximations as above, yields

$$[D_t D_t u = c^2 D_x D_x + f]_i^n \,. \tag{22}$$

Writing this out and solving for the unknown u_i^{n+1} results in

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + C^2(u_{i+1}^n - 2u_i^n + u_{i-1}^n) + \Delta t^2 f_i^n \,. \tag{23}$$

The equation for the first time step must be rederived. The discretization of the initial condition $u_t = V(x)$ at $t = 0$ becomes

$$[D_{2t} u = V]_i^0 \quad \Rightarrow \quad u_i^{-1} = u_i^1 - 2\Delta t V_i,$$

which, when inserted in (23) for $n = 0$, gives the special formula

$$u_i^1 = u_i^0 - \Delta t V_i + \frac{1}{2} C^2 \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) + \frac{1}{2}\Delta t^2 f_i^n \,. \tag{24}$$

2.2 Using an analytical solution of physical significance

Many wave problems feature sinusoidal oscillations in time and space. For example, the original PDE problem (1)-(5) allows a solution

$$u_{\mathrm{e}}(x,y,t)) = A \sin\left(\frac{\pi}{L}x\right) \cos\left(\frac{\pi}{L}ct\right) \,. \tag{25}$$

This u_{e} fulfills the PDE with $f = 0$, boundary conditions $u_{\mathrm{e}}(0,t) = u_{\mathrm{e}}(L,0) = 0$, as well as initial conditions $I(x) = A \sin\left(\frac{\pi}{L}x\right)$ and $V = 0$.

It is common to use such exact solutions of physical interest to verify implementations. However, the numerical solution u_i^n will only be an approximation

to $u_{\rm e}(x_i, t_n)$. We no have knowledge of the precise size of the error in this approximation, and therefore we can never know if discrepancies between the computed u_i^n and $u_{\rm e}(x_i, t_n)$ are caused by mathematical approximations or programming errors. In particular, if a plot of the computed solution u_i^n and the exact one (25) looks similar, many are attempted to claim that the implementation works, but there can still be serious programming errors although color plots look nice.

The only way to use exact physical solutions like (25) for serious and thorough verification is to run a series of finer and finer meshes, measure the integrated error in each mesh, and from this information estimate the convergence rate. If these rates are very close to 2, we have strong evidence that the implementation works.

2.3 Manufactured solution

One problem with the exact solution (25) is that it requires a simplification ($V = 0, f = 0$) of the implemented problem (17)-(21). An advantage of using a manufactured solution is that we can test all terms in the PDE problem. The idea of this approach is to set up some chosen solution and fit the source term, boundary conditions, and initial conditions to be compatible with the chosen solution. Given that our boundary conditions in the implementation are $u(0, t) = u(L, t) = 0$, we must choose a solution that fulfills these conditions. One example is

$$u_{\rm e}(x, t) = x(L - x)\sin t\,.$$

Inserted in the PDE $u_{tt} = c^2 u_{xx} + f$ we get

$$-x(L - x)\sin t = -2\sin t + f \quad \Rightarrow f = (2 - x(L - x))\sin t\,.$$

The initial conditions become

$$\begin{aligned} u(x, 0) =& I(x) = 0, \\ u_t(x, 0) =& V(x) = -x(L - x)\,. \end{aligned}$$

To verify the code, we run a series of refined meshes and compute the convergence rates. Such tests rely on an assumption that some measure E of the numerical error is related to the discretization parameters through

$$E = C_t \Delta t^r + C_x \Delta x^p,$$

where C_t, C_x, r, and p are constants. The constants r and p are known as the *convergence rates* in time and space, respectively. From the accuracy in the finite difference approximations, we expect $r = p = 2$. This is confirmed by truncation error analysis and other types of analysis. By using an exact solution of the PDE problem, we can empirically compute the error measure E on a sequence of refined meshes and see if the rates $r = p = 2$ are obtained. We will not be concerned with estimating the constants C_t and C_x.

It is advantageous to introduce a single discretization parameter $h = \Delta t = \hat{c}\Delta x$ for some constant $\hat{c}$ (the idea is to keep $\Delta t^r/\Delta x^p$ constant). Since Δt and Δx are related through the Courant number, $\Delta t = C\Delta x/c$, we set $h = \Delta t$, and then $\Delta x = hc/C$. Now the expression for the error measure is greatly simplified:

$$E = C_t\Delta t^r + C_x\Delta x^r = C_t h^r + \frac{C_x c}{C}h^r = \hat{C}h^r, \quad \hat{C} = C_t + \frac{C_x c}{C}\,.$$

We choose an initial discretization parameter h_0 and run experiments with decreasing h: $h_i = 2^{-i}h_0$, $i = 1, 2, \ldots, m$. Halving h in each experiment is not necessary, but a common choice. For each experiment we must record E and h. A standard choice of error measure is the ℓ^2 or ℓ^∞ norm of the error mesh function e_i^n:

$$E = ||e_i^n||_{\ell^2} = \left(\Delta t\Delta x\sum_{n=0}^{N_t}\sum_{i=0}^{N_x}(e_i^n)^2\right)^{\frac{1}{2}}, \quad e_i^n = u_{\rm e}(x_i, t_n) - u_i^n, \tag{26}$$

$$E = ||e_i^n||_{\ell^\infty} = \max_{i,n}|e_n^i|\,. \tag{27}$$

In Python, one can compute $\sum_i(e_i^{n+1})^2$ at each time step and accumulate the value in some sum variable, say `e2_sum`. At the final time step one can do `sqrt(dt*dx*e2_sum)`. For the ℓ^∞ norm one must compare the maximum error at a time level (`e.max()`) with the global maximum over the time domain: `e_max = max(e_max, e.max())`.

An alternative error measure is to use a spatial norm at one time step only, e.g., the end time T:

$$E = ||e_i^n||_{\ell^2} = \left(\Delta x\sum_{i=0}^{N_x}(e_i^n)^2\right)^{\frac{1}{2}}, \quad e_i^n = u_{\rm e}(x_i, t_n) - u_i^n, \tag{28}$$

$$E = ||e_i^n||_{\ell^\infty} = \max_{0\leq i\leq N_x}|e_n^i|\,. \tag{29}$$

Let E_i be the error measure in experiment (mesh) number i and let h_i be the corresponding discretization parameter (h). With the error model $E_i = \hat{C}h_i^r$, we can estimate r by comparing two consecutive experiments: $E_{i+1} = \hat{C}h_{i+1}^r$ and $E_i = \hat{C}h_i^r$. Dividing the two equations eliminates $\hat{C}$ and solving for r_i yields

$$r_i = \frac{\ln E_{i+1}/E_i}{\ln h_{i+1}/h_i}, \quad i = 0, \ldots, m-1\,.$$

We should for the present discretization method observe that r_i approaches 2 as i increases.

2.4 Constructing an exact solution of the discrete equations

With a manufactured or known analytical solution, as outlined above, we can estimate convergence rates and see if they have the correct asymptotic behavior. Experience shows that this is a quite good verification technique in that many common bugs will destroy the convergence rates. A significantly better test would be to check that the numerical solution is exactly what it should be. This will in general require knowledge of the numerical error, which we do not have. However, it is possible to look for solutions where we can show that the numerical error vanishes, i.e., the solution of the PDE problem is also a solution of the discrete equations. This property often arises if the exact solution is a lower-order polynomial. (Truncation error analysis leads to error measures that involve derivatives of the exact solution. In the present problem, the truncation error involves 4th-order derivatives of u in space and time. Choosing u as a polynomial of degree three or less will therefore lead to vanishing error.)

We shall now illustrate the construction of an exact solution of the PDE problem and the discrete equations. Our choice of manufactured solution is quadratic in space and linear in time. More specifically, we set

$$u_{\rm e}(x,t) = x(L-x)(1+\frac{1}{2}t), \tag{30}$$

which by insertion in the PDE leads to $f(x,t) = 2(1+t)c^2$. This $u_{\rm e}$ fulfills the boundary conditions $u=0$ and demands $I(x) = x(L-x)$ and $V(x) = \frac{1}{2}x(L-x)$.

To realize that the chosen $u_{\rm e}$ is that it is also an exact solution of the discrete equations, we first establish the results

$$[D_tD_tt^2]^n = \frac{t_{n+1}^2 - 2t_n^2 + t_{n-1}^2}{\Delta t^2} = (n+1)^2 - n^2 + (n-1)^2 = 2, \tag{31}$$

$$[D_tD_tt]^n = \frac{t_{n+1} - 2t_n + t_{n-1}}{\Delta t^2} = \frac{((n+1) - n + (n-1))\Delta t}{\Delta t^2} = 0\,. \tag{32}$$

Hence,

$$[D_tD_tu_{\rm e}]_i^n = x_i(L-x_i)[D_tD_t(1+\frac{1}{2}t)]^n = x_i(L-x_i)\frac{1}{2}[D_tD_tt]^n = 0,$$

and

$$\begin{aligned}[D_xD_xu_{\rm e}]_i^n &= (1+\frac{1}{2}t_n)[D_xD_x(xL-x^2)]_i = (1+\frac{1}{2}t_n)[LD_xD_xx - D_xD_xx^2]_i \\ &= -2(1+\frac{1}{2}t_n)\,.\end{aligned}$$

Now, $f_i^n = 2(1+\frac{1}{2}t_n)c^2$ and we get

$$[D_tD_tu_{\rm e} - c^2D_xD_xu_{\rm e} - f]_i^n = 0 - c^2(-1)2(1+\frac{1}{2}t_n + 2(1+\frac{1}{2}t_n)c^2 = 0\,.$$

Moreover, $u_{\rm e}(x_i,0) = I(x_i)$, $\partial u_{\rm e}/\partial t = V(x_i)$ at $t=0$, and $u_{\rm e}(x_0,t) = u_{\rm e}(x_{N_x},0) = 0$. Also the modified scheme for the first time step is fulfilled by $u_{\rm e}(x_i,t_n)$.

Therefore, the exact solution $u_{\rm e}(x,t) = x(L-x)(1+t/2)$ of the PDE problem is also an exact solution of the discrete problem. We can use this result to check that the computed u_i^n vales from an implementation equals $u_{\rm e}(x_i,t_n)$ within machine precision, *regardless of the mesh spacings* Δx and Δt! Nevertheless, there might be stability restrictions on Δx and Δt, so the test can only be run for a mesh that is compatible with the stability criterion (which in the present case is $C \leq 1$, to be derived later).

Notice.

A product of quadratic or linear expressions in the various independent variables, as shown above, will often fulfill both the continuous and discrete PDE problem and can therefore be very useful solutions for verifying implementations. However, for 1D wave equations of the type $u_t = c^2 u_{xx}$ we shall see that there is always another much more powerful way of generating exact solutions (just set $C=1$).

3 Implementation

This section present the complete computational algorithm, its implementation in Python code, animation of the solution, and verification of the implementation.

A real implementation of the basic computational algorithm from Sections 1.5 and 1.6 can be encapsulated in a function, taking all the input data for the problem as arguments. The physical input data consists of c, $I(x)$, $V(x)$, $f(x,t)$, L, and T. The numerical input is the mesh parameters Δt and Δx.

Instead of specifying Δt *and* Δx, we can specify one of them and the Courant number C instead, since having explicit control of the Courant number is convenient when investigating the numerical method. Many find it natural to prescribe the resolution of the spatial grid and set N_x. The solver function can then compute $\Delta t = CL/(cN_x)$. However, for comparing $u(x,t)$ curves (as functions of x) for various Courant numbers, especially in animations in time, it is more convenient to keep Δt fixed for all C and let Δx vary according to $\Delta x = c\Delta t/C$. (With Δt fixed, all frames correspond to the same time t, and plotting curves with different spatial resolution is trivial.)

The solution at all spatial points at a new time level is stored in an array `u` (of length N_x+1). We need to decide what do to with this solution, e.g., visualize the curve, analyze the values, or write the array to file for later use. The decision what to do is left to the user in a suppled function

```
def user_action(u, x, t, n):
```

where `u` is the solution at the spatial points `x` at time `t[n]`.

3.1 Making a solver function

A first attempt at a solver function is listed below.

```
from numpy import *

def solver(I, V, f, c, L, dt, C, T, user_action=None):
    """Solve u_tt=c^2*u_xx + f on (0,L)x(0,T]."""
    Nt = int(round(T/dt))
    t = linspace(0, Nt*dt, Nt+1)   # Mesh points in time
    dx = dt*c/float(C)
    Nx = int(round(L/dx))
    x = linspace(0, L, Nx+1)       # Mesh points in space
    C2 = C**2                      # Help variable in the scheme
    if f is None or f == 0 :
        f = lambda x, t: 0
    if V is None or V == 0:
        V = lambda x: 0

    u   = zeros(Nx+1)   # Solution array at new time level
    u_1 = zeros(Nx+1)   # Solution at 1 time level back
    u_2 = zeros(Nx+1)   # Solution at 2 time levels back

    import time;  t0 = time.clock()  # for measuring CPU time

    # Load initial condition into u_1
    for i in range(0,Nx+1):
        u_1[i] = I(x[i])

    if user_action is not None:
        user_action(u_1, x, t, 0)

    # Special formula for first time step
    n = 0
    for i in range(1, Nx):
        u[i] = u_1[i] + dt*V(x[i]) + \
               0.5*C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
               0.5*dt**2*f(x[i], t[n])
    u[0] = 0;  u[Nx] = 0

    if user_action is not None:
        user_action(u, x, t, 1)

    # Switch variables before next step
    u_2[:] = u_1;  u_1[:] = u

    for n in range(1, Nt):
        # Update all inner points at time t[n+1]
        for i in range(1, Nx):
            u[i] = - u_2[i] + 2*u_1[i] + \
                     C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
                     dt**2*f(x[i], t[n])

        # Insert boundary conditions
```

```
        u[0] = 0;  u[Nx] = 0
        if user_action is not None:
            if user_action(u, x, t, n+1):
                break

        # Switch variables before next step
        u_2[:] = u_1;  u_1[:] = u

    cpu_time = t0 - time.clock()
    return u, x, t, cpu_time
```

3.2 Verification: exact quadratic solution

We use the test problem derived in Section 2.1 for verification. Here is a function realizing this verification as a nose test:

```
import nose.tools as nt

def test_quadratic():
    """Check that u(x,t)=x(L-x)(1+t/2) is exactly reproduced."""
    def u_exact(x, t):
        return x*(L-x)*(1 + 0.5*t)

    def I(x):
        return u_exact(x, 0)

    def V(x):
        return 0.5*u_exact(x, 0)

    def f(x, t):
        return 2*(1 + 0.5*t)*c**2

    L = 2.5
    c = 1.5
    C = 0.75
    Nx = 3  # Very coarse mesh for this exact test
    dt = C*(L/Nx)/c
    T = 18

    u, x, t, cpu = solver(I, V, f, c, L, dt, C, T)
    u_e = u_exact(x, t[-1])
    diff = abs(u - u_e).max()
    nt.assert_almost_equal(diff, 0, places=14)
```

3.3 Visualization: animating the solution

Now that we have verified the implementation it is time to do a real computation where we also display the evolution of the waves on the screen.

Visualization via SciTools. The following `viz` function defines a `user_action` callback function for plotting the solution at each time level:

```
def viz(I, V, f, c, L, dt, C, T, umin, umax, animate=True):
    """Run solver and visualize u at each time level."""
    import scitools.std as plt
    import time, glob, os

    def plot_u(u, x, t, n):
        """user_action function for solver."""
        plt.plot(x, u, 'r-',
                 xlabel='x', ylabel='u',
                 axis=[0, L, umin, umax],
                 title='t=%f' % t[n], show=True)
        # Let the initial condition stay on the screen for 2
        # seconds, else insert a pause of 0.2 s between each plot
        time.sleep(2) if t[n] == 0 else time.sleep(0.2)
        plt.savefig('frame_%04d.png' % n)  # for movie making

    # Clean up old movie frames
    for filename in glob.glob('frame_*.png'):
        os.remove(filename)

    user_action = plot_u if animate else None
    u, x, t, cpu = solver(I, V, f, c, L, dt, C, T, user_action)

    # Make movie files
    fps = 4  # Frames per second
    plt.movie('frame_*.png', encoder='html', fps=fps,
              output_file='movie.html')
    codec2ext = dict(flv='flv', libx264='mp4', libvpx='webm',
                     libtheora='ogg')
    filespec = 'frame_%04d.png'
    movie_program = 'avconv'  # or 'ffmpeg'
    for codec in codec2ext:
        ext = codec2ext[codec]
        cmd = '%(movie_program)s -r %(fps)d -i %(filespec)s '\
              '-vcodec %(codec)s movie.%(ext)s' % vars()
        os.system(cmd)
```

A function inside another function, like `plot_u` in the above code segment, has access to *and remembers* all the local variables in the surrounding code inside the `viz` function (!). This is known in computer science as a *closure* and is very convenient to program with. For example, the `plt` and `time` modules defined outside `plot_u` are accessible for `plot_u` when the function is called (as `user_action`) in the `solver` function. Some may think, however, that a class instead of a closure is a cleaner and easier-to-understand implementation of the user action function, see Section 8.

Making movie files. Several hardcopies of the animation are made from the `frame_*.png` files. We use the `avconv` (or `ffmpeg`) programs to combine individual plot files to movies in modern formats: Flash, MP4, Webm, and Ogg. A typical `avconv` (or `ffmpeg`) command for creating a movie file in Ogg format with 4 frames per second built from a collection of plot files with names generated by `frame_%04d.png`, look like

```
Terminal> avconv -r 4 -i frame_%04d.png -c:v libtheora movie.ogg
```

The different formats require different video encoders (`-c:v`) to be installed: Flash applies `flv`, WebM applies `libvpx`, and MP4 applies `libx264`:

```
Terminal> avconv -r 4 -i frame_%04d.png -c:v flv movie.flv
Terminal> avconv -r 4 -i frame_%04d.png -c:v libvpx movie.webm
Terminal> avconv -r 4 -i frame_%04d.png -c:v libx264 movie.mp4
```

Players like `vlc`, `mplayer`, `gxine`, and `totem` can be used to play these movie files.

Note that padding the frame counter with zeros in the `frame_*.png` files, as specified by the `%04d` format, is essential so that the wildcard notation `frame_*.png` expands to the correct set of files.

The `plt.movie` function also creates a `movie.html` file with a movie player for displaying the `frame_*.png` files in a web browser. This movie player can be generated from the command line too

```
Terminal> scitools movie encoder=html output_file=movie.html \
          fps=4 frame_*.png
```

Skipping frames for animation speed. Sometimes the time step is small and T is large, leading to an inconveniently large number of plot files and a slow animation on the screen. The solution to such a problem is to decide on a total number of frames in the animation, `num_frames`, and plot the solution only at every `every` frame. The total number of time levels (i.e., maximum possible number of frames) is the length of `t`, `t.size`, and if we want `num_frames`, we need to plot every `t.size/num_frames` frame:

```
every = int(t.size/float(num_frames))
if n % every == 0 or n == t.size-1:
    st.plot(x, u, 'r-', ...)
```

The initial condition (`n=0`) is natural to include, and as `n % every == 0` will very seldom be true for the very final frame, we also ensure that `n == t.size-1` and hence the final frame is included.

A simple choice of numbers may illustrate the formulas: say we have 801 frames in total (`t.size`) and we allow only 60 frames to be plotted. Then we need to plot every 801/60 frame, which with integer division yields 13 as `every`. Using the mod function, `n % every`, this operation is zero every time `n` can be divided by 13 without a remainder. That is, the `if` test is true when `n` equals $0, 13, 26, 39, ..., 780, 801$. The associated code is included in the `plot_u` function in the file `wave1D_u0v.py`.

Visualization via Matplotlib. The previous code based on the `plot` interface from `scitools.std` can be run with Matplotlib as the visualization backend, but if one desires to program directly with Matplotlib, quite different code is needed. Matplotlib's interactive mode must be turned on:

```
import matplotlib.pyplot as plt
plt.ion()  # interactive mode on
```

The most commonly used animation technique with Matplotlib is to update the data in the plot at each time level:

```
# Make a first plot
lines = plt.plot(t, u)
# call plt.axis, plt.xlabel, plt.ylabel, etc. as desired

# At later time levels
lines[0].set_ydata(u)
plt.legend('t=%g' % t[n])
plt.draw()  # make updated plot
plt.savefig(...)
```

An alternative is to rebuild the plot at every time level:

```
plt.clf()        # delete any previous curve(s)
plt.axis([...])
plt.plot(t, u)
# plt.xlabel, plt.legend and other decorations
plt.draw()
plt.savefig(...)
```

Many prefer to work with figure and axis objects as in MATLAB:

```
fig = plt.figure()
...
fig.clf()
ax = fig.gca()
ax.axis(...)
ax.plot(t, u)
# ax.set_xlabel, ax.legend and other decorations
plt.draw()
fig.savefig(...)
```

3.4 Running a case

The first demo of our 1D wave equation solver concerns vibrations of a string that is initially deformed to a triangular shape, like when picking a guitar string:

$$I(x) = \begin{cases} ax/x_0, & x < x_0, \\ a(L-x)/(L-x_0), & \text{otherwise} \end{cases} \tag{33}$$

We choose $L = 75$ cm, $x_0 = 0.8L$, $a = 5$ mm, and a time frequency $\nu = 440$ Hz. The relation between the wave speed c and ν is $c = \nu\lambda$, where λ is the

wavelength, taken as $2L$ because the longest wave on the string form half a wavelength. There is no external force, so $f = 0$, and the string is at rest initially so that $V = 0$.

Regarding numerical parameters, we need to specify a Δt. Sometimes it is more natural to think of a spatial resolution instead of a time step. A natural semi-coarse spatial resolution in the present problem is $N_x = 50$. We can then choose the associated Δt (as required by the `viz` and `solver` functions) as the stability limit: $\Delta t = L/(N_x c)$. This is the Δt to be specified, but notice that if $C < 1$, the actual Δx computed in `solver` gets larger than L/N_x: $\Delta x = c\Delta t/C = L/(N_x C)$. (The reason is that we fix Δt and adjust Δx, so if C gets smaller, the code implements this effect in terms of a larger Δx.)

A function for setting the physical and numerical parameters and calling `viz` in this application goes as follows:

```
def guitar(C):
    """Triangular wave (pulled guitar string)."""
    L = 0.75
    x0 = 0.8*L
    a = 0.005
    freq = 440
    wavelength = 2*L
    c = freq*wavelength
    omega = 2*pi*freq
    num_periods = 1
    T = 2*pi/omega*num_periods
    # Choose dt the same as the stability limit for Nx=50
    dt = L/50./c

    def I(x):
        return a*x/x0 if x < x0 else a/(L-x0)*(L-x)

    umin = -1.2*a;  umax = -umin
    cpu = viz(I, 0, 0, c, L, dt, C, T, umin, umax, animate=True)
```

The associated program has the name `wave1D_u0.py`. Run the program and watch the movie of the vibrating string.

3.5 The benefits of scaling

The previous example demonstrated that quite some work is needed with establishing relevant physical parameters for a case. By *scaling* the mathematical problem we can often reduce the need to estimate physical parameters dramatically. A scaling consists of introducing new independent and dependent variables, with the aim that the absolute value of these vary between 0 and 1:

$$\bar{x} = \frac{x}{L}, \quad \bar{t} = \frac{c}{L}t, \quad \bar{u} = \frac{u}{a}\,.$$

Replacing old by new variables in the PDE, using $f = 0$, and dropping the bars, results in the *scaled equation* $u_{tt} = u_{xx}$. This equation has no physical parameter (!).

If we have a program implemented for the physical wave equation with dimensions, we can obtain the dimensionless, scaled version by setting $c = 1$. The initial condition corresponds to (185), but with setting $a = 1$, $L = 1$, and $x_0 \in [0, 1]$. This means that we only need to decide on the x_0 value as a fraction of unity, because the scaled problem corresponds to setting all other parameters to unity! In the code we can just set `a=c=L=1`, `x0=0.8`, and there is no need to calculate with wavelengths and frequencies to estimate c.

The only non-trivial parameter to estimate in the scaled problem is the final end time of the simulation, or more precisely, how it relates to periods in periodic solutions in time, since we often want to express the end time as a certain number of periods. Suppose as u behaves as $\sin(\omega t)$ in time in variables with dimension. The corresponding period is $P = 2\pi/\omega$. The frequency ω is related to the wavelength λ of the waves through the relations $\omega = kc$ and $k = 2\pi/\lambda$, giving $\omega = 2\pi c/\lambda$ and $P = \lambda/c$. It remains to estimate λ. With $u(x,t) = F(x)\sin\omega t$ we find from $u_{tt} = c^2 u_{xx}$ that $c^2 F'' + \omega^2 F = 0$, and the boundary conditions demand $F(0) = F(L) = 0$. The solution is $F(x) = \sin(x\pi/L)$, which has wavelength $\lambda = 2\pi/(\pi/L) = 2L$. One period is therefore given by $P = 2L/c$. The dimensionless period is $\bar{P} = Pc/L = 2$.

4 Vectorization

The computational algorithm for solving the wave equation visits one mesh point at a time and evaluates a formula for the new value u_i^{n+1} at that point. Technically, this is implemented by a loop over array elements in a program. Such loops may run slowly in Python (and similar interpreted languages such as R and MATLAB). One technique for speeding up loops is to perform operations on entire arrays instead of working with one element at a time. This is referred to as *vectorization*, *vector computing*, or *array computing*. Operations on whole arrays are possible if the computations involving each element is independent of each other and therefore can, at least in principle, be performed simultaneously. Vectorization not only speeds up the code on serial computers, but it also makes it easy to exploit parallel computing.

4.1 Operations on slices of arrays

Efficient computing with `numpy` arrays demands that we avoid loops and compute with entire arrays at once (or at least large portions of them). Consider this calculation of differences $d_i = u_{i+1} - u_i$:

```
n = u.size
for i in range(0, n-1):
    d[i] = u[i+1] - u[i]
```

All the differences here are independent of each other. The computation of `d` can therefore alternatively be done by subtracting the array $(u_0, u_1, \ldots, u_{n-1})$ from the array where the elements are shifted one index upwards: $(u_1, u_2, \ldots, u_n)$,

see Figure 3. The former subset of the array can be expressed by `u[0:n-1]`, `u[0:-1]`, or just `u[:-1]`, meaning from index 0 up to, but not including, the last element (`-1`). The latter subset is obtained by `u[1:n]` or `u[1:]`, meaning from index 1 and the rest of the array. The computation of `d` can now be done without an explicit Python loop:

```
d = u[1:] - u[:-1]
```

or with explicit limits if desired:

```
d = u[1:n] - u[0:n-1]
```

Indices with a colon, going from an index to (but not including) another index are called *slices*. With `numpy` arrays, the computations are still done by loops, but in efficient, compiled, highly optimized code in C or Fortran. Such array operations can also easily be distributed among many processors on parallel computers. We say that the *scalar code* above, working on an element (a scalar) at a time, has been replaced by an equivalent *vectorized code*. The process of vectorizing code is called *vectorization*.

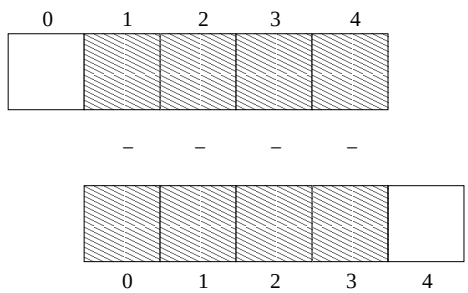

Figure 3: Illustration of subtracting two slices of two arrays.

Test the understanding.

Newcomers to vectorization are encouraged to choose a small array `u`, say with five elements, and simulate with pen and paper both the loop version and the vectorized version.

Finite difference schemes basically contains differences between array elements with shifted indices. Consider the updating formula

```
for i in range(1, n-1):
    u2[i] = u[i-1] - 2*u[i] + u[i+1]
```

The vectorization consists of replacing the loop by arithmetics on slices of arrays of length `n-2`:

```
u2 = u[:-2] - 2*u[1:-1] + u[2:]
u2 = u[0:n-2] - 2*u[1:n-1] + u[2:n]   # alternative
```

Note that `u2` here gets length `n-2`. If `u2` is already an array of length `n` and we want to use the formula to update all the "inner" elements of `u2`, as we will when solving a 1D wave equation, we can write

```
u2[1:-1]  = u[:-2] - 2*u[1:-1] + u[2:]
u2[1:n-1] = u[0:n-2] - 2*u[1:n-1] + u[2:n]   # alternative
```

Pen and paper calculations with a small array will demonstrate what is actually going on. The expression on the right-hand side are done in the following steps, involving temporary arrays with intermediate results, since we can only work with two arrays at a time in arithmetic expressions:

```
temp1 = 2*u[1:-1]
temp2 = u[0:-2] - temp1
temp3 = temp2 + u[2:]
u2[1:-1] = temp3
```

We can extend the previous example to a formula with an additional term computed by calling a function:

```
def f(x):
    return x**2 + 1

for i in range(1, n-1):
    u2[i] = u[i-1] - 2*u[i] + u[i+1] + f(x[i])
```

Assuming `u2`, `u`, and `x` all have length `n`, the vectorized version becomes

```
u2[1:-1] = u[:-2] - 2*u[1:-1] + u[2:] + f(x[1:-1])
```

4.2 Finite difference schemes expressed as slices

We now have the necessary tools to vectorize the algorithm for the wave equation. There are three loops: one for the initial condition, one for the first time step, and finally the loop that is repeated for all subsequent time levels. Since only the latter is repeated a potentially large number of times, we limit the efforts of vectorizing the code to this loop:

```
for i in range(1, Nx):
    u[i] = 2*u_1[i] - u_2[i] + \
           C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])
```

The vectorized version becomes

```
u[1:-1] = - u_2[1:-1] + 2*u_1[1:-1] + \
          C2*(u_1[:-2] - 2*u_1[1:-1] + u_1[2:])
```

or

```
u[1:Nx] = 2*u_1[1:Nx]- u_2[1:Nx] + \
          C2*(u_1[0:Nx-1] - 2*u_1[1:Nx] + u_1[2:Nx+1])
```

The program `wave1D_u0v.py` contains a new version of the function `solver` where both the scalar and the vectorized loops are included (the argument `version` is set to `scalar` or `vectorized`, respectively).

4.3 Verification

We may reuse the quadratic solution $u_{\rm e}(x,t) = x(L-x)(1+\frac{1}{2}t)$ for verifying also the vectorized code. A nose test can now test both the scalar and the vectorized version. Moreover, we may use a `user_action` function that compares the computed and exact solution at each time level and performs a test:

```
def test_quadratic():
    """
    Check the scalar and vectorized versions work for
    a quadratic u(x,t)=x(L-x)(1+t/2) that is exactly reproduced.
    """
    # The following function must work for x as array or scalar
    u_exact = lambda x, t: x*(L - x)*(1 + 0.5*t)
    I = lambda x: u_exact(x, 0)
    V = lambda x: 0.5*u_exact(x, 0)
    # f is a scalar (zeros_like(x) works for scalar x too)
    f = lambda x, t: zeros_like(x) + 2*c**2*(1 + 0.5*t)

    L = 2.5
    c = 1.5
    C = 0.75
    Nx = 3  # Very coarse mesh for this exact test
    dt = C*(L/Nx)/c
    T = 18

    def assert_no_error(u, x, t, n):
        u_e = u_exact(x, t[n])
        diff = abs(u - u_e).max()
        nt.assert_almost_equal(diff, 0, places=13)

    solver(I, V, f, c, L, dt, C, T,
           user_action=assert_no_error, version='scalar')
    solver(I, V, f, c, L, dt, C, T,
           user_action=assert_no_error, version='vectorized')
```

Lambda functions.

The code segment above demonstrates how to achieve very compact code with the use of lambda functions for the various input parameters that require a Python function. In essence,

```
f = lambda x, t: L*(x-t)**2
```

is equivalent to

```
def f(x, t):
    return L(x-t)**2
```

Note that lambda functions can just contain a single expression and no statements.

One advantage with lambda functions is that they can be used directly in calls:

```
solver(I=lambda x: sin(pi*x/L), V=0, f=0, ...)
```

4.4 Efficiency measurements

Running the `wave1D_u0v.py` code with the previous string vibration example for $N_x = 50, 100, 200, 400, 800$, and measuring the CPU time (see the `run_efficiency_experiments` function), shows that the vectorized code runs substantially faster: the scalar code uses approximately a factor $N_x/5$ more time!

4.5 Remark on the updating of arrays

At the end of each time step we need to update the `u_2` and `u_1` arrays such that they have the right content for the next time step:

```
u_2[:] = u_1
u_1[:] = u
```

The order here is important! (Updating `u_1` first, makes `u_2` equal to `u`, which is wrong.)

The assignment `u_1[:] = u` copies the content of the `u` array into the elements of the `u_1` array. Such copying takes time, but little compared to computing `u` from the finite difference formula, even when the formula has a vectorized implementation. However, efficiency of program code is a key topic when solving PDEs numerically, so it must be mentioned that there exists a much more efficient way of making the arrays `u_2` and `u_1` ready for the next time step. The idea is based on *switching references* and explained below.

A Python variable is actually a reference to some object (C programmers may think of pointers). Instead of copying data, we can let `u_2` refer to the `u_1` object and `u_1` refer to the `u` object. A naive implementation like

```
u_2 = u_1
u_1 = u
```

will fail, however, because now `u_1` and `u` refers to the same object and the update of `u` from the finite difference formula at the next time step will overwrite `u_1` and lead to erroneous computations. Also, with the suggested change of references, the reference to the `u_2` array is lost, and we have in fact only two arrays. The solution to this problem is to ensure that `u` points to the `u_2` array. This is mathematically wrong, but new correct values will be filled into `u` at the next time step.

The correct switch of references is then

```
tmp = u_2
u_2 = u_1
u_1 = u
u = tmp
```

We can get rid of the temporary reference `tmp` by writing

```
u_2, u_1, u = u_1, u, u_2
```

This update will be used in later implementations.

Caution:

The update `u_2, u_1, u = u_1, u, u_2` leaves wrong content in `u` at the final time step. This means that if we return `u`, as we do in the example codes here, we actually return `u_2`, which is obviously wrong. It is therefore important to adjust the content of `u` to `u = u_1` before returning `u`.

5 Exercises

Exercise 1: Simulate a standing wave

The purpose of this exercise is to simulate standing waves on $[0, L]$ and illustrate the error in the simulation. Standing waves arise from an initial condition

$$u(x, 0) = A \sin\left(\frac{\pi}{L} m x\right),$$

where m is an integer and A is a freely chosen amplitude. The corresponding exact solution can be computed and reads

$$u_{\mathrm{e}}(x, t) = A \sin\left(\frac{\pi}{L} m x\right) \cos\left(\frac{\pi}{L} m c t\right).$$

a) Explain that for a function $\sin kx \cos \omega t$ the wave length in space is $\lambda = 2\pi/k$ and the period in time is $P = 2\pi/\omega$. Use these expressions to find the wave length in space and period in time of u_{e} above.

b) Import the `solver` function `wave1D_u0.py` into a new file where the `viz` function is reimplemented such that it plots either the numerical *and* the exact solution, *or* the error.

c) Make animations where you illustrate how the error $e_i^n = u_{\rm e}(x_i, t_n) - u_i^n$ develops and increases in time. Also make animations of u and $u_{\rm e}$ simultaneously.

Hint 1. Quite long time simulations are needed in order to display significant discrepancies between the numerical and exact solution.

Hint 2. A possible set of parameters is $L = 12$, $m = 9$, $c = 2$, $A = 1$, $N_x = 80$, $C = 0.8$. The error mesh function e^n can be simulated for 10 periods, while 20-30 periods are needed to show significant differences between the curves for the numerical and exact solution.
Filename: `wave_standing.py`.

Remarks. The important parameters for numerical quality are C and $k\Delta x$, where $C = c\Delta t/\Delta x$ is the Courant number and k is defined above ($k\Delta x$ is proportional to how many mesh points we have per wave length in space, see Section 10.4 for explanation).

Exercise 2: Add storage of solution in a user action function

Extend the `plot_u` function in the file `wave1D_u0.py` to also store the solutions `u` in a list. To this end, declare `all_u` as an empty list in the `viz` function, outside `plot_u`, and perform an append operation inside the `plot_u` function. Note that a function, like `plot_u`, inside another function, like `viz`, remembers all local variables in `viz` function, including `all_u`, even when `plot_u` is called (as `user_action`) in the `solver` function. Test both `all_u.append(u)` and `all_u.append(u.copy())`. Why does one of these constructions fail to store the solution correctly? Let the `viz` function return the `all_u` list converted to a two-dimensional `numpy` array. Filename: `wave1D_u0_s_store.py`.

Exercise 3: Use a class for the user action function

Redo Exercise 2 using a class for the user action function. That is, define a class `Action` where the `all_u` list is an attribute, and implement the user action function as a method (the special method `__call__` is a natural choice). The class versions avoids that the user action function depends on parameters defined outside the function (such as `all_u` in Exercise 2). Filename: `wave1D_u0_s2c.py`.

Exercise 4: Compare several Courant numbers in one movie

The goal of this exercise is to make movies where several curves, corresponding to different Courant numbers, are visualized. Import the `solver` function from the `wave1D_u0_s` movie in a new file `wave_compare.py`. Reimplement the `viz`

function such that it can take a list of `C` values as argument and create a movie with solutions corresponding to the given `C` values. The `plot_u` function must be changed to store the solution in an array (see Exercise 2 or 3 for details), `solver` must be computed for each value of the Courant number, and finally one must run through each time step and plot all the spatial solution curves in one figure and store it in a file.

The challenge in such a visualization is to ensure that the curves in one plot corresponds to the same time point. The easiest remedy is to keep the time and space resolution constant and change the wave velocity c to change the Courant number. Filename: `wave_numerics_comparison.py`.

Project 5: Calculus with 1D mesh functions

This project explores integration and differentiation of mesh functions, both with scalar and vectorized implementations. We are given a mesh function f_i on a spatial one-dimensional mesh $x_i = i\Delta x$, $i = 0, \ldots, N_x$, over the interval $[a, b]$.

a) Define the discrete derivative of f_i by using centered differences at internal mesh points and one-sided differences at the end points. Implement a scalar version of the computation in a Python function and supply a nose test for the linear case $f(x) = 4x - 2.5$ where the discrete derivative should be exact.

b) Vectorize the implementation of the discrete derivative. Extend the nose test to check the validity of the implementation.

c) To compute the discrete integral F_i of f_i, we assume that the mesh function f_i varies linearly between the mesh points. Let $f(x)$ be such a linear interpolant of f_i. We then have

$$F_i = \int_{x_0}^{x_i} f(x)dx\,.$$

The exact integral of a piecewise linear function $f(x)$ is given by the Trapezoidal rule. S how that if F_i is already computed, we can find F_{i+1} from

$$F_{i+1} = F_i + \frac{1}{2}(f_i + f_{i+1})\Delta x\,.$$

Make a function for a scalar implementation of the discrete integral as a mesh function. That is, the function should return F_i for $i = 0, \ldots, N_x$. For a nose test one can use the fact that the above defined discrete integral of a linear function (say $f(x) = 4x - 2.5$) is exact.

d) Vectorize the implementation of the discrete integral. Extend the nose test to check the validity of the implementation.

Hint. Interpret the recursive formula for F_{i+1} as a sum. Make an array with each element of the sum and use the "cumsum" (`numpy.cumsum`) operation to compute the accumulative sum: `numpy.cumsum([1,3,5])` is `[1,4,9]`.

e) Create a class `MeshCalculus` that can integrate and differentiate mesh functions. The class can just define some methods that call the previously implemented Python functions. Here is an example on the usage:

```
import numpy as np
calc = MeshCalculus(vectorized=True)
x = np.linspace(0, 1, 11)        # mesh
f = np.exp(x)                    # mesh function
df = calc.differentiate(f, x)    # discrete derivative
F = calc.integrate(f, x)         # discrete anti-derivative
```

Filename: `mesh_calculus_1D.py`.

6 Generalization: reflecting boundaries

The boundary condition $u = 0$ makes u change sign at the boundary, while the condition $u_x = 0$ perfectly reflects the wave, see a web page or a movie file for demonstration. Our next task is to explain how to implement the boundary condition $u_x = 0$, which is more complicated to express numerically and also to implement than a given value of u. We shall present two methods for implementing $u_x = 0$ in a finite difference scheme, one based on deriving a modified stencil at the boundary, and another one based on extending the mesh with ghost cells and ghost points.

6.1 Neumann boundary condition

When a wave hits a boundary and is to be reflected back, one applies the condition

$$\frac{\partial u}{\partial n} \equiv \boldsymbol{n}\cdot\nabla u = 0\,. \tag{34}$$

The derivative $\partial/\partial n$ is in the outward normal direction from a general boundary. For a 1D domain $[0, L]$, we have that

$$\left.\frac{\partial}{\partial n}\right|_{x=L} = \frac{\partial}{\partial x},\quad \left.\frac{\partial}{\partial n}\right|_{x=0} = -\frac{\partial}{\partial x}\,.$$

Boundary condition terminology.

Boundary conditions that specify the value of $\partial u/\partial n$, or shorter u_n, are known as Neumann conditions, while Dirichlet conditions refer to specifications of u. When the values are zero ($\partial u/\partial n = 0$ or $u = 0$) we speak about *homogeneous* Neumann or Dirichlet conditions.

6.2 Discretization of derivatives at the boundary

How can we incorporate the condition (34) in the finite difference scheme? Since we have used central differences in all the other approximations to derivatives in the scheme, it is tempting to implement (34) at $x = 0$ and $t = t_n$ by the difference

$$\frac{u^n_{-1} - u^n_1}{2\Delta x} = 0\,. \tag{35}$$

The problem is that u^n_{-1} is not a u value that is being computed since the point is outside the mesh. However, if we combine (35) with the scheme for $i = 0$,

$$u^{n+1}_i = -u^{n-1}_i + 2u^n_i + C^2\left(u^n_{i+1} - 2u^n_i + u^n_{i-1}\right), \tag{36}$$

we can eliminate the fictitious value u^n_{-1}. We see that $u^n_{-1} = u^n_1$ from (35), which can be used in (36) to arrive at a modified scheme for the boundary point u^{n+1}_0:

$$u^{n+1}_i = -u^{n-1}_i + 2u^n_i + 2C^2\left(u^n_{i+1} - u^n_i\right), \quad i = 0\,. \tag{37}$$

Figure 4 visualizes this equation for computing u^3_0 in terms of u^2_0, u^1_0, and u^2_1.

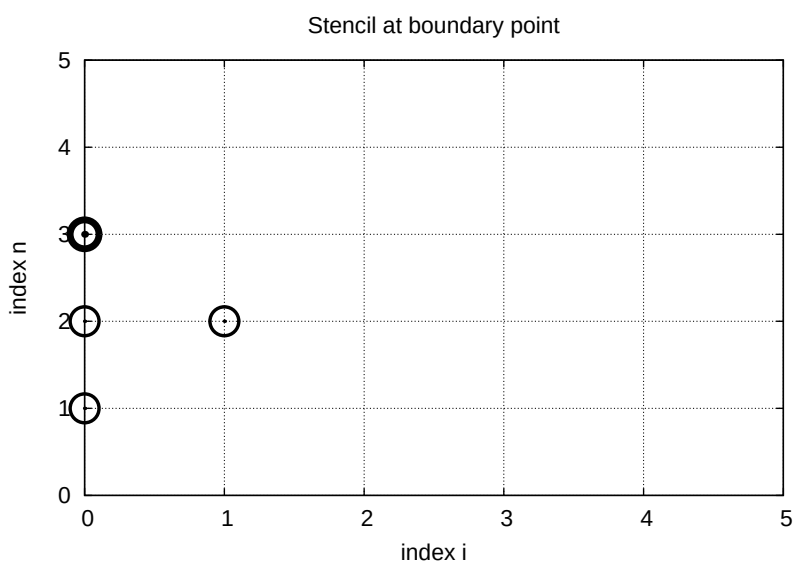

Figure 4: Modified stencil at a boundary with a Neumann condition.

Similarly, (34) applied at $x = L$ is discretized by a central difference

$$\frac{u^n_{N_x+1} - u^n_{N_x-1}}{2\Delta x} = 0\,. \tag{38}$$

Combined with the scheme for $i = N_x$ we get a modified scheme for the boundary value $u^{n+1}_{N_x}$:

$$u_i^{n+1} = -u_i^{n-1} + 2u_i^n + 2C^2\left(u_{i-1}^n - u_i^n\right), \quad i = N_x\,. \tag{39}$$

The modification of the scheme at the boundary is also required for the special formula for the first time step. How the stencil moves through the mesh and is modified at the boundary can be illustrated by an animation in a web page or a movie file.

6.3 Implementation of Neumann conditions

The implementation of the special formulas for the boundary points can benefit from using the general formula for the interior points also at the boundaries, but replacing u_{i-1}^n by u_{i+1}^n when computing u_i^{n+1} for $i = 0$ and u_{i+1}^n by u_{i-1}^n for $i = N_x$. This is achieved by just replacing the index $i-1$ by $i+1$ for $i = 0$ and $i+1$ by $i-1$ for $i = N_x$. In a program, we introduce variables to hold the value of the offset indices: `im1` for `i-1` and `ip1` for `i+1`. It is now just a manner of defining `im1` and `ip1` properly for the internal points and the boundary points. The coding for the latter reads

```
i = 0
ip1 = i+1
im1 = ip1  # i-1 -> i+1
u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])

i = Nx
im1 = i-1
ip1 = im1  # i+1 -> i-1
u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])
```

We can in fact create one loop over both the internal and boundary points and use only one updating formula:

```
for i in range(0, Nx+1):
    ip1 = i+1 if i < Nx else i-1
    im1 = i-1 if i > 0  else i+1
    u[i] = u_1[i] + C2*(u_1[im1] - 2*u_1[i] + u_1[ip1])
```

The program `wave1D_n0.py` contains a complete implementation of the 1D wave equation with boundary conditions $u_x = 0$ at $x = 0$ and $x = L$.

It would be nice to modify the `test_quadratic` test case from the `wave1D_u0.py` with Dirichlet conditions, described in Section 4.3. However, the Neumann conditions requires the polynomial variation in x directory to be of third degree, which causes challenging problems with designing a test where the numerical solution is known exactly. Exercise 10 outlines ideas and code for this purpose. The only test in `wave1D_n0.py` is to start with a plug wave at rest and see that the initial condition is reached again perfectly after one period of motion, if $C = 1$.

6.4 Index set notation

We shall introduce a special notation for index sets, consisting of writing x_i, $i \in \mathcal{I}_x$, instead of $i = 0, \ldots, N_x$. Obviously, $\mathcal{I}_x$ must be the set $\mathcal{I}_x = \{0, \ldots, N_x\}$, but it is often advantageous to have a symbol for this set rather than specifying all its elements. This saves writing and makes specification of algorithms and implementation of computer code easier.

The first index in the set will be denoted $\mathcal{I}_x^0$ and the last $\mathcal{I}_x^{-1}$. Sometimes we need to count from the second element in the set, and the notation $\mathcal{I}_x^+$ is then used. Correspondingly, $\mathcal{I}_x^-$ means $\{0, \ldots, N_x - 1\}$. All the indices corresponding to inner grid points are $\mathcal{I}_x^i = \{1, \ldots, N_x - 1\}$. For the time domain we find it natural to explicitly use 0 as the first index, so we will usually write $n = 0$ and t_0 rather than $n = \mathcal{I}_t^0$. We also avoid notation like $x_{\mathcal{I}_x^{-1}}$ and will instead use x_i, $i = \mathcal{I}_x^{-1}$.

The Python code associated with index sets applies the following conventions:

Notation	Python
$\mathcal{I}_x$	`Ix`
$\mathcal{I}_x^0$	`Ix[0]`
$\mathcal{I}_x^{-1}$	`Ix[-1]`
$\mathcal{I}_x^-$	`Ix[:-1]`
$\mathcal{I}_x^+$	`Ix[1:]`
$\mathcal{I}_x^i$	`Ix[1:-1]`

An important feature of the index set notation is that it keeps our formulas and code independent of how we count mesh points. For example, the notation $i \in \mathcal{I}_x$ or $i = \mathcal{I}_x^0$ remains the same whether $\mathcal{I}_x$ is defined as above or as starting at 1, i.e., $\mathcal{I}_x = \{1, \ldots, Q\}$. Similarly, we can in the code define `Ix=range(Nx+1)` or `Ix=range(1,Q)`, and expressions like `Ix[0]` and `Ix[1:-1]` remain correct. One application where the index set notation is convenient is conversion of code from a language where arrays has base index 0 (e.g., Python and C) to languages where the base index is 1 (e.g., MATLAB and Fortran). Another important application is implementation of Neumann conditions via ghost points (see next section).

For the current problem setting in the x, t plane, we work with the index sets

$$\mathcal{I}_x = \{0, \ldots, N_x\}, \quad \mathcal{I}_t = \{0, \ldots, N_t\}, \tag{40}$$

defined in Python as

```
Ix = range(0, Nx+1)
It = range(0, Nt+1)
```

A finite difference scheme can with the index set notation be specified as

$$
\begin{aligned}
u_i^{n+1} &= -u_i^{n-1} + 2u_i^n + C^2\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right), \quad i\in\mathcal{I}_x^i,\ n\in\mathcal{I}_t^i,\\
u_i &= 0, \quad i=\mathcal{I}_x^0,\ n\in\mathcal{I}_t^i,\\
u_i &= 0, \quad i=\mathcal{I}_x^{-1},\ n\in\mathcal{I}_t^i,
\end{aligned}
$$

and implemented by code like

```
for n in It[1:-1]:
    for i in Ix[1:-1]:
        u[i] = - u_2[i] + 2*u_1[i] + \
                C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])
    i = Ix[0];  u[i] = 0
    i = Ix[-1]; u[i] = 0
```

Notice.

The program `wave1D_dn.py` applies the index set notation and solves the 1D wave equation $u_{tt} = c^2u_{xx} + f(x,t)$ with quite general boundary and initial conditions:

- $x=0$: $u=U_0(t)$ or $u_x=0$
- $x=L$: $u=U_L(t)$ or $u_x=0$
- $t=0$: $u=I(x)$
- $t=0$: $u_t=I(x)$

The program combines Dirichlet and Neumann conditions, scalar and vectorized implementation of schemes, and the index notation into one piece of code. A lot of test examples are also included in the program:

- A rectangular plug profile as initial condition (easy to use as test example as the rectangle should jump one cell per time step when $C=1$, without any numerical errors).
- A Gaussian function as initial condition.
- A triangular profile as initial condition, which resembles the typical initial shape of a guitar string.
- A sinusoidal variation of u at $x=0$ and either $u=0$ or $u_x=0$ at $x=L$.
- An exact analytical solution $u(x,t) = \cos(m\pi t/L)\sin(\frac{1}{2}m\pi x/L)$, which can be used for convergence rate tests.

6.5 Alternative implementation via ghost cells

Idea. Instead of modifying the scheme at the boundary, we can introduce extra points outside the domain such that the fictitious values u^n_{-1} and $u^n_{N_x+1}$ are defined in the mesh. Adding the intervals $[-\Delta x, 0]$ and $[L, L+\Delta x]$, often referred to as *ghost cells*, to the mesh gives us all the needed mesh points, corresponding to $i = -1, 0, \ldots, N_x, N_x+1$. The extra points $i = -1$ and $i = N_x+1$ are known as *ghost points*, and values at these points, u^n_{-1} and $u^n_{N_x+1}$, are called *ghost values*.

The important idea is to ensure that we always have

$$u^n_{-1} = u^n_1 \text{ and } u^n_{N_x+1} = u^n_{N_x-1},$$

because then the application of the standard scheme at a boundary point $i = 0$ or $i = N_x$ will be correct and guarantee that the solution is compatible with the boundary condition $u_x = 0$.

Implementation. The `u` array now needs extra elements corresponding to the ghost cells and points. Two new point values are needed:

```
u   = zeros(Nx+3)
```

The arrays `u_1` and `u_2` must be defined accordingly.

Unfortunately, a major indexing problem arises with ghost cells. The reason is that Python indices *must* start at 0 and `u[-1]` will always mean the last element in `u`. This fact gives, apparently, a mismatch between the mathematical indices $i = -1, 0, \ldots, N_x+1$ and the Python indices running over `u`: `0,..,Nx+2`. One remedy is to change the mathematical notation of the scheme, as in

$$u^{n+1}_i = \cdots, \quad i = 1, \ldots, N_x+1,$$

meaning that the ghost points correspond to $i = 0$ and $i = N_x+1$. A better solution is to use the ideas of Section 6.4: we hide the specific index value in an index set and operate with inner and boundary points using the index set notation.

To this end, we define `u` with proper length and `Ix` to be the corresponding indices for the real physical points $(1, 2, \ldots, N_x+1)$:

```
u = zeros(Nx+3)
Ix = range(1, u.shape[0]-1)
```

That is, the boundary points have indices `Ix[0]` and `Ix[-1]` (as before). We first update the solution at all physical mesh points (i.e., interior points in the mesh extended with ghost cells):

```
for i in Ix:
    u[i] = - u_2[i] + 2*u_1[i] + \
           C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])
```

It remains to update the ghost points. For a boundary condition $u_x = 0$, the ghost value must equal to the value at the associated inner mesh point. Computer code makes this statement precise:

```
i = Ix[0]           # x=0 boundary
u[i-1] = u[i+1]
i = Ix[-1]          # x=L boundary
u[i+1] = u[i-1]
```

The physical solution to be plotted is now in `u[1:-1]`, or equivalently `u[Ix[0]:Ix[-1]+1]`, so this slice is the quantity to be returned from a solver function. A complete implementation appears in the program `wave1D_n0_ghost.py`.

Warning.

We have to be careful with how the spatial and temporal mesh points are stored. Say we let `x` be the physical mesh points,

```
x = linspace(0, L, Nx+1)
```

"Standard coding" of the initial condition,

```
for i in Ix:
    u_1[i] = I(x[i])
```

becomes wrong, since `u_1` and `x` have different lengths and the index `i` corresponds to two different mesh points. In fact, `x[i]` corresponds to `u[1+i]`. A correct implementation is

```
for i in Ix:
    u_1[i] = I(x[i-Ix[0]])
```

Similarly, a source term usually coded as `f(x[i], t[n])` is incorrect if `x` is defined to be the physical points, so `x[i]` must be replaced by `x[i-Ix[0]]`.

An alternative remedy is to let `x` also cover the ghost points such that `u[i]` is the value at `x[i]`.

The ghost cell is only added to the boundary where we have a Neumann condition. Suppose we have a Dirichlet condition at $x = L$ and a homogeneous Neumann condition at $x = 0$. One ghost cell $[-\Delta x, 0]$ is added to the mesh, so the index set for the physical points becomes $\{1, \ldots, N_x + 1\}$. A relevant implementation is

```
u = zeros(Nx+2)
Ix = range(1, u.shape[0])
...
for i in Ix[:-1]:
    u[i] = - u_2[i] + 2*u_1[i] + \
           C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1]) + \
           dt2*f(x[i-Ix[0]], t[n])
i = Ix[-1]
u[i] = U_0       # set Dirichlet value
i = Ix[0]
u[i-1] = u[i+1]  # update ghost value
```

The physical solution to be plotted is now in `u[1:]` or (as always) `u[Ix[0]:Ix[-1]+1]`.

7 Generalization: variable wave velocity

Our next generalization of the 1D wave equation (1) or (17) is to allow for a variable wave velocity c: $c = c(x)$, usually motivated by wave motion in a domain composed of different physical media with different properties for propagating waves and hence different wave velocities c. Figure

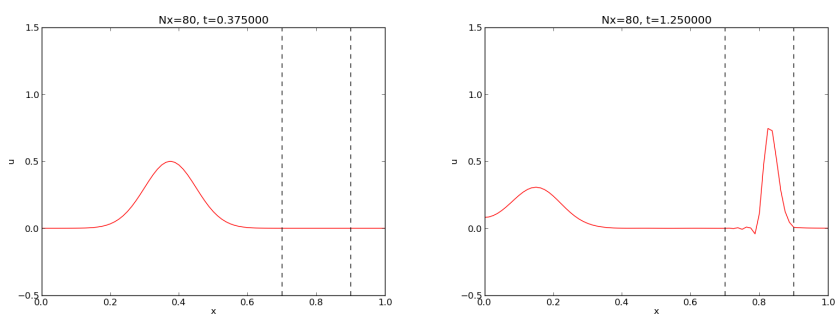

Figure 5: Left: wave entering another medium; right: transmitted and reflected wave .

7.1 The model PDE with a variable coefficient

Instead of working with the squared quantity $c^2(x)$ we shall for notational convenience introduce $q(x) = c^2(x)$. A 1D wave equation with variable wave velocity often takes the form

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right) + f(x,t)\,. \tag{41}$$

This equation sampled at a mesh point (x_i, t_n) reads

$$\frac{\partial^2}{\partial t^2}u(x_i,t_n) = \frac{\partial}{\partial x}\left(q(x_i)\frac{\partial}{\partial x}u(x_i,t_n)\right) + f(x_i,t_n),$$

where the only new term is

$$\frac{\partial}{\partial x}\left(q(x_i)\frac{\partial}{\partial x}u(x_i,t_n)\right) = \left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n .$$

7.2 Discretizing the variable coefficient

The principal idea is to first discretize the outer derivative. Define

$$\phi = q(x)\frac{\partial u}{\partial x},$$

and use a centered derivative around $x = x_i$ for the derivative of ϕ:

$$\left[\frac{\partial \phi}{\partial x}\right]_i^n \approx \frac{\phi_{i+\frac{1}{2}} - \phi_{i-\frac{1}{2}}}{\Delta x} = [D_x\phi]_i^n .$$

Then discretize

$$\phi_{i+\frac{1}{2}} = q_{i+\frac{1}{2}}\left[\frac{\partial u}{\partial x}\right]_{i+\frac{1}{2}}^n \approx q_{i+\frac{1}{2}}\frac{u_{i+1}^n - u_i^n}{\Delta x} = [qD_xu]_{i+\frac{1}{2}}^n .$$

Similarly,

$$\phi_{i-\frac{1}{2}} = q_{i-\frac{1}{2}}\left[\frac{\partial u}{\partial x}\right]_{i-\frac{1}{2}}^n \approx q_{i-\frac{1}{2}}\frac{u_i^n - u_{i-1}^n}{\Delta x} = [qD_xu]_{i-\frac{1}{2}}^n .$$

These intermediate results are now combined to

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n \approx \frac{1}{\Delta x^2}\left(q_{i+\frac{1}{2}}\left(u_{i+1}^n - u_i^n\right) - q_{i-\frac{1}{2}}\left(u_i^n - u_{i-1}^n\right)\right) . \tag{42}$$

With operator notation we can write the discretization as

$$\left[\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)\right]_i^n \approx [D_xqD_xu]_i^n . \tag{43}$$

Remark.

Many are tempted to use the chain rule on the term $\frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right)$, but this is not a good idea when discretizing such a term.

7.3 Computing the coefficient between mesh points

If q is a known function of x, we can easily evaluate $q_{i+\frac{1}{2}}$ simply as $q(x_{i+\frac{1}{2}})$ with $x_{i+\frac{1}{2}} = x_i + \frac{1}{2}\Delta x$. However, in many cases c, and hence q, is only known as a

discrete function, often at the mesh points x_i. Evaluating q between two mesh points x_i and x_{i+1} can then be done by averaging in three ways:

$$q_{i+\frac{1}{2}} \approx \frac{1}{2}\left(q_i + q_{i+1}\right) = [\overline{q}^x]_i, \qquad \text{(arithmetic mean)} \tag{44}$$

$$q_{i+\frac{1}{2}} \approx 2\left(\frac{1}{q_i} + \frac{1}{q_{i+1}}\right)^{-1}, \qquad \text{(harmonic mean)} \tag{45}$$

$$q_{i+\frac{1}{2}} \approx \left(q_i q_{i+1}\right)^{1/2}, \qquad \text{(geometric mean)} \tag{46}$$

The arithmetic mean in (44) is by far the most commonly used averaging technique.

With the operator notation from (44) we can specify the discretization of the complete variable-coefficient wave equation in a compact way:

$$[D_t D_t u = D_x \overline{q}^x D_x u + f]_i^n \,. \tag{47}$$

From this notation we immediately see what kind of differences that each term is approximated with. The notation $\overline{q}^x$ also specifies that the variable coefficient is approximated by an arithmetic mean, the definition being $[\overline{q}^x]_{i+\frac{1}{2}} = (q_i + q_{i+1})/2$. With the notation $[D_x q D_x u]_i^n$, we specify that q is evaluated directly, as a function, between the mesh points: $q(x_{i-\frac{1}{2}})$ and $q(x_{i+\frac{1}{2}})$.

Before any implementation, it remains to solve (47) with respect to u_i^{n+1}:

$$\begin{aligned} u_i^{n+1} = & -u_i^{n-1} + 2u_i^n + \\ & \left(\frac{\Delta x}{\Delta t}\right)^2 \left(\frac{1}{2}(q_i + q_{i+1})(u_{i+1}^n - u_i^n) - \frac{1}{2}(q_i + q_{i-1})(u_i^n - u_{i-1}^n)\right) + \\ & \Delta t^2 f_i^n \,. \end{aligned} \tag{48}$$

7.4 How a variable coefficient affects the stability

The stability criterion derived in Section 10.3 reads $\Delta t \leq \Delta x/c$. If $c = c(x)$, the criterion will depend on the spatial location. We must therefore choose a Δt that is small enough such that no mesh cell has $\Delta x/c(x) > \Delta t$. That is, we must use the largest c value in the criterion:

$$\Delta t \leq \beta \frac{\Delta x}{\max_{x\in[0,L]} c(x)} \,. \tag{49}$$

The parameter β is included as a safety factor: in some problems with a significantly varying c it turns out that one must choose $\beta < 1$ to have stable solutions ($\beta = 0.9$ may act as an all-round value).

7.5 Neumann condition and a variable coefficient

Consider a Neumann condition $\partial u/\partial x = 0$ at $x = L = N_x\Delta x$, discretized as

$$\frac{u^n_{i+1} - u^n_{i-1}}{2\Delta x} = 0 \quad u^n_{i+1} = u^n_{i-1},$$

for $i = N_x$. Using the scheme (48) at the end point $i = N_x$ with $u^n_{i+1} = u^n_{i-1}$ results in

$$\begin{aligned}
u^{n+1}_i &= -u^{n-1}_i + 2u^n_i + \\
&\quad \left(\frac{\Delta x}{\Delta t}\right)^2 \left(q_{i+\frac{1}{2}}(u^n_{i-1} - u^n_i) - q_{i-\frac{1}{2}}(u^n_i - u^n_{i-1})\right) + \\
&\quad \Delta t^2 f^n_i && (50)\\
&= -u^{n-1}_i + 2u^n_i + \left(\frac{\Delta x}{\Delta t}\right)^2 (q_{i+\frac{1}{2}} + q_{i-\frac{1}{2}})(u^n_{i-1} - u^n_i) + \Delta t^2 f^n_i && (51)\\
&\approx -u^{n-1}_i + 2u^n_i + \left(\frac{\Delta x}{\Delta t}\right)^2 2q_i(u^n_{i-1} - u^n_i) + \Delta t^2 f^n_i\,. && (52)
\end{aligned}$$

Here we used the approximation

$$\begin{aligned}
q_{i+\frac{1}{2}} + q_{i-\frac{1}{2}} &= q_i + \left(\frac{dq}{dx}\right)_i \Delta x + \left(\frac{d^2q}{dx^2}\right)_i \Delta x^2 + \cdots + \\
&\quad q_i - \left(\frac{dq}{dx}\right)_i \Delta x + \left(\frac{d^2q}{dx^2}\right)_i \Delta x^2 + \cdots \\
&= 2q_i + 2\left(\frac{d^2q}{dx^2}\right)_i \Delta x^2 + \mathcal{O}(\Delta x^4) \\
&\approx 2q_i\,. && (53)
\end{aligned}$$

An alternative derivation may apply the arithmetic mean of q in (48), leading to the term

$$(q_i + \frac{1}{2}(q_{i+1} + q_{i-1}))(u^n_{i-1} - u^n_i)\,.$$

Since $\frac{1}{2}(q_{i+1} + q_{i-1}) = q_i + \mathcal{O}(\Delta x^2)$, we end up with $2q_i(u^n_{i-1} - u^n_i)$ for $i = N_x$ as we did above.

A common technique in implementations of $\partial u/\partial x = 0$ boundary conditions is to assume $dq/dx = 0$ as well. This implies $q_{i+1} = q_{i-1}$ and $q_{i+1/2} = q_{i-1/2}$ for $i = N_x$. The implications for the scheme are

$$
\begin{aligned}
u_i^{n+1} &= -u_i^{n-1} + 2u_i^n + \\
&\left(\frac{\Delta x}{\Delta t}\right)^2 \left(q_{i+\frac{1}{2}}(u_{i-1}^n - u_i^n) - q_{i-\frac{1}{2}}(u_i^n - u_{i-1}^n)\right) + \\
&\Delta t^2 f_i^n \qquad (54)\\
&= -u_i^{n-1} + 2u_i^n + \left(\frac{\Delta x}{\Delta t}\right)^2 2q_{i-\frac{1}{2}}(u_{i-1}^n - u_i^n) + \Delta t^2 f_i^n\,. \qquad (55)
\end{aligned}
$$

7.6 Implementation of variable coefficients

The implementation of the scheme with a variable wave velocity may assume that c is available as an array `c[i]` at the spatial mesh points. The following loop is a straightforward implementation of the scheme (48):

```
for i in range(1, Nx):
    u[i] = - u_2[i] + 2*u_1[i] + \
           C2*(0.5*(q[i] + q[i+1])*(u_1[i+1] - u_1[i])  - \
               0.5*(q[i] + q[i-1])*(u_1[i] - u_1[i-1])) + \
           dt2*f(x[i], t[n])
```

The coefficient `C2` is now defined as `(dt/dx)**2` and *not* as the squared Courant number since the wave velocity is variable and appears inside the parenthesis.

With Neumann conditions $u_x = 0$ at the boundary, we need to combine this scheme with the discrete version of the boundary condition, as shown in Section 7.5. Nevertheless, it would be convenient to reuse the formula for the interior points and just modify the indices `ip1=i+1` and `im1=i-1` as we did in Section 6.3. Assuming $dq/dx = 0$ at the boundaries, we can implement the scheme at the boundary with the following code.

```
i = 0
ip1 = i+1
im1 = ip1
u[i] = - u_2[i] + 2*u_1[i] + \
       C2*(0.5*(q[i] + q[ip1])*(u_1[ip1] - u_1[i])  - \
           0.5*(q[i] + q[im1])*(u_1[i] - u_1[im1])) + \
       dt2*f(x[i], t[n])
```

With ghost cells we can just reuse the formula for the interior points also at the boundary, provided that the ghost values of both u and q are correctly updated to ensure $u_x = 0$ and $q_x = 0$.

A vectorized version of the scheme with a variable coefficient at internal points in the mesh becomes

```
u[1:-1] = - u_2[1:-1] + 2*u_1[1:-1] + \
          C2*(0.5*(q[1:-1] + q[2:])*(u_1[2:] - u_1[1:-1]) -
              0.5*(q[1:-1] + q[:-2])*(u_1[1:-1] - u_1[:-2])) + \
          dt2*f(x[1:-1], t[n])
```

7.7 A more general model PDE with variable coefficients

Sometimes a wave PDE has a variable coefficient also in front of the time-derivative term:

$$\varrho(x)\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x)\frac{\partial u}{\partial x}\right) + f(x,t)\,. \tag{56}$$

A natural scheme is

$$[\varrho D_t D_t u = D_x \overline{q}^x D_x u + f]^n_i\,. \tag{57}$$

We realize that the ϱ coefficient poses no particular difficulty because the only value ϱ^n_i enters the formula above (when written out). There is hence no need for any averaging of ϱ. Often, ϱ will be moved to the right-hand side, also without any difficulty:

$$[D_t D_t u = \varrho^{-1} D_x \overline{q}^x D_x u + f]^n_i\,. \tag{58}$$

7.8 Generalization: damping

Waves die out by two mechanisms. In 2D and 3D the energy of the wave spreads out in space, and energy conservation then requires the amplitude to decrease. This effect is not present in 1D. Damping is another cause of amplitude reduction. For example, the vibrations of a string die out because of damping due to air resistance and non-elastic effects in the string.

The simplest way of including damping is to add a first-order derivative to the equation (in the same way as friction forces enter a vibrating mechanical system):

$$\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = c^2\frac{\partial^2 u}{\partial x^2} + f(x,t), \tag{59}$$

where $b \geq 0$ is a prescribed damping coefficient.

A typical discretization of (59) in terms of centered differences reads

$$[D_t D_t u + bD_{2t} u = c^2 D_x D_x u + f]^n_i\,. \tag{60}$$

Writing out the equation and solving for the unknown u^{n+1}_i gives the scheme

$$u^{n+1}_i = (1+\frac{1}{2}b\Delta t)^{-1}((\frac{1}{2}b\Delta t - 1)u^{n-1}_i + 2u^n_i + C^2\left(u^n_{i+1} - 2u^n_i + u^n_{i-1}\right) + \Delta t^2 f^n_i), \tag{61}$$

for $i \in \mathcal{I}^i_x$ and $n \geq 1$. New equations must be derived for u^1_i, and for boundary points in case of Neumann conditions.

The damping is very small in many wave phenomena and then only evident for very long time simulations. This makes the standard wave equation without damping relevant for a lot of applications.

8 Building a general 1D wave equation solver

The program wave1D_dn_vc.py is a fairly general code for 1D wave propagation problems that targets the following initial-boundary value problem

$$
\begin{aligned}
u_t &= (c^2(x)u_x)_x + f(x,t), & x\in(0,L),\ t\in(0,T] &\quad (62)\\
u(x,0) &= I(x), & x\in[0,L] &\quad (63)\\
u_t(x,0) &= V(t), & x\in[0,L] &\quad (64)\\
u(0,t) &= U_0(t) \text{ or } u_x(0,t)=0, & t\in(0,T] &\quad (65)\\
u(L,t) &= U_L(t) \text{ or } u_x(L,t)=0, & t\in(0,T] &\quad (66)
\end{aligned}
$$

The solver function is a natural extension of the simplest solver function in the initial wave1D_u0.py program, extended with Neumann boundary conditions ($u_x = 0$), a possibly time-varying boundary condition on u ($U_0(t)$, $U_L(t)$), and a variable wave velocity. The different code segments needed to make these extensions are shown and commented upon in the preceding text.

The vectorization is only applied inside the time loop, not for the initial condition or the first time steps, since this initial work is negligible for long time simulations in 1D problems.

The following sections explain various more advanced programming techniques applied in the general 1D wave equation solver.

8.1 User action function as a class

A useful feature in the wave1D_dn_vc.py program is the specification of the user_action function as a class. Although the plot_u function in the viz function of previous wave1D*.py programs remembers the local variables in the viz function, it is a cleaner solution to store the needed variables together with the function, which is exactly what a class offers.

A class for flexible plotting, cleaning up files, and making a movie files like function viz and plot_u did can be coded as follows:

```
class PlotSolution:
    """
    Class for the user_action function in solver.
    Visualizes the solution only.
    """
    def __init__(self,
                 casename='tmp',    # Prefix in filenames
                 umin=-1, umax=1,   # Fixed range of y axis
                 pause_between_frames=None,  # Movie speed
                 backend='matplotlib',       # or 'gnuplot'
                 screen_movie=True, # Show movie on screen?
                 title='',          # Extra message in title
                 every_frame=1):    # Show every_frame frame
        self.casename = casename
        self.yaxis = [umin, umax]
        self.pause = pause_between_frames
        module = 'scitools.easyviz.' + backend + '_'
```

```
        exec('import %s as plt' % module)
        self.plt = plt
        self.screen_movie = screen_movie
        self.title = title
        self.every_frame = every_frame

        # Clean up old movie frames
        for filename in glob('frame_*.png'):
            os.remove(filename)

    def __call__(self, u, x, t, n):
        if n % self.every_frame != 0:
            return
        title = 't=%.3g' % t[n]
        if self.title:
            title = self.title + ' ' + title
        self.plt.plot(x, u, 'r-',
                      xlabel='x', ylabel='u',
                      axis=[x[0], x[-1],
                            self.yaxis[0], self.yaxis[1]],
                      title=title,
                      show=self.screen_movie)
        # pause
        if t[n] == 0:
            time.sleep(2)  # let initial condition stay 2 s
        else:
            if self.pause is None:
                pause = 0.2 if u.size < 100 else 0
            time.sleep(pause)

        self.plt.savefig('%s_frame_%04d.png' % (self.casename, n))
```

Understanding this class requires quite some familiarity with Python in general and class programming in particular.

The constructor shows how we can flexibly import the plotting engine as (typically) `scitools.easyviz.gnuplot_` or `scitools.easyviz.matplotlib_` (note the trailing underscore). With the `screen_movie` parameter we can suppress displaying each movie frame on the screen. Alternatively, for slow movies associated with fine meshes, one can set `every_frame` to, e.g., 10, causing every 10 frames to be shown.

The `__call__` method makes `PlotSolution` instances behave like functions, so we can just pass an instance, say `p`, as the `user_action` argument in the `solver` function, and any call to `user_action` will be a call to `p.__call__`.

8.2 Pulse propagation in two media

The function `pulse` in `wave1D_dn_vc.py` demonstrates wave motion in heterogeneous media where c varies. One can specify an interval where the wave velocity is decreased by a factor `slowness_factor` (or increased by making this factor less than one). Four types of initial conditions are available: a rectangular pulse (`plug`), a Gaussian function (`gaussian`), a "cosine hat" consisting of one period of the cosine function (`cosinehat`), and half a period of a "cosine hat" (`half-cosinehat`). These peak-shaped initial conditions can be placed in the

middle (`loc='center'`) or at the left end (`loc='left'`) of the domain. The `pulse` function is a flexible tool for playing around with various wave shapes and location of a medium with a different wave velocity:

```
def pulse(C=1, Nx=200, animate=True, version='vectorized', T=2,
          loc='center', pulse_tp='gaussian', slowness_factor=2,
          medium=[0.7, 0.9], every_frame=1, sigma=0.05):
    """
    Various peaked-shaped initial conditions on [0,1].
    Wave velocity is decreased by the slowness_factor inside
    medium. The loc parameter can be 'center' or 'left',
    depending on where the initial pulse is to be located.
    The sigma parameter governs the width of the pulse.
    """
    # Use scaled parameters: L=1 for domain length, c_0=1
    # for wave velocity outside the domain.
    L = 1.0
    c_0 = 1.0
    if loc == 'center':
        xc = L/2
    elif loc == 'left':
        xc = 0

    if pulse_tp in ('gaussian','Gaussian'):
        def I(x):
            return exp(-0.5*((x-xc)/sigma)**2)
    elif pulse_tp == 'plug':
        def I(x):
            return 0 if abs(x-xc) > sigma else 1
    elif pulse_tp == 'cosinehat':
        def I(x):
            # One period of a cosine
            w = 2
            a = w*sigma
            return 0.5*(1 + cos(pi*(x-xc)/a)) \
                   if xc - a <= x <= xc + a else 0

    elif pulse_tp == 'half-cosinehat':
        def I(x):
            # Half a period of a cosine
            w = 4
            a = w*sigma
            return cos(pi*(x-xc)/a) \
                   if xc - 0.5*a <= x <= xc + 0.5*a else 0
    else:
        raise ValueError('Wrong pulse_tp="%s"' % pulse_tp)

    def c(x):
        return c_0/slowness_factor \
               if medium[0] <= x <= medium[1] else c_0

    umin=-0.5; umax=1.5*I(xc)
    casename = '%s_Nx%s_sf%s' % \
               (pulse_tp, Nx, slowness_factor)
    action = PlotMediumAndSolution(
        medium, casename=casename, umin=umin, umax=umax,
        every_frame=every_frame, screen_movie=animate)

    dt = (L/Nx)/c  # choose the stability limit with given Nx
    # Lower C will then use this dt, but smaller Nx
```

```
    solver(I=I, V=None, f=None, c=c, U_0=None, U_L=None,
           L=L, dt=dt, C=C, T=T,
           user_action=action, version=version,
           stability_safety_factor=1)
```

The `PlotMediumAndSolution` class used here is a subclass of `PlotSolution` where the medium with reduced c value, as specified by the `medium` interval, is visualized in the plots.

Notice.

The argument N_x in the `pulse` function does not correspond to the actual spatial resolution of $C < 1$, since the `solver` function takes a fixed Δt and C, and adjusts Δx accordingly. As seen in the `pulse` function, the specified Δt is chosen according to the limit $C = 1$, so if $C < 1$, Δt remains the same, but the `solver` function operates with a larger Δx and smaller N_x than was specified in the call to `pulse`. The practical reason is that we always want to keep Δt fixed such that plot frames and movies are synchronized in time regardless of the value of C (i.e., Δx is varies when the Courant number varies).

The reader is encouraged to play around with the `pulse` function:

```
>>> import wave1D_dn_vc as w
>>> w.pulse(loc='left', pulse_tp='cosinehat', Nx=50, every_frame=10)
```

To easily kill the graphics by Ctrl-C and restart a new simulation it might be easier to run the above two statements from the command line with

```
Terminal> python -c 'import wave1D_dn_vc as w; w.pulse(...)'
```

9 Exercises

Exercise 6: Find the analytical solution to a damped wave equation

Consider the wave equation with damping (59). The goal is to find an exact solution to a wave problem with damping. A starting point is the standing wave solution from Exercise 1. It becomes necessary to include a damping term e^{-ct} and also have both a sine and cosine component in time:

$$u_{\rm e}(x,t) = e^{-\beta t}\sin kx\left(A\cos\omega t + B\sin\omega t\right)\ .$$

Find k from the boundary conditions $u(0,t) = u(L,t) = 0$. Then use the PDE to find constraints on β, ω, A, and B. Set up a complete initial-boundary value problem and its solution. Filename: `damped_waves.pdf`.

Problem 7: Explore symmetry boundary conditions

Consider the simple "plug" wave where $\Omega = [-L, L]$ and

$$I(x) = \begin{cases} 1, & x \in [-\delta, \delta], \\ 0, & \text{otherwise} \end{cases}$$

for some number $0 < \delta < L$. The other initial condition is $u_t(x, 0) = 0$ and there is no source term f. The boundary conditions can be set to $u = 0$. The solution to this problem is symmetric around $x = 0$. This means that we can simulate the wave process in only the half of the domain $[0, L]$.

a) Argue why the symmetry boundary condition is $u_x = 0$ at $x = 0$.

Hint. Symmetry of a function about $x = x_0$ means that $f(x_0 + h) = f(x_0 - h)$.

b) Perform simulations of the complete wave problem from on $[-L, L]$. Thereafter, utilize the symmetry of the solution and run a simulation in half of the domain $[0, L]$, using a boundary condition at $x = 0$. Compare the two solutions and make sure that they are the same.

c) Prove the symmetry property of the solution by setting up the complete initial-boundary value problem and showing that if $u(x, t)$ is a solution, then also $u(-x, t)$ is a solution.
Filename: `wave1D_symmetric`.

Exercise 8: Send pulse waves through a layered medium

Use the `pulse` function in `wave1D_dn_vc.py` to investigate sending a pulse, located with its peak at $x = 0$, through the medium to the right where it hits another medium for $x \in [0.7, 0.9]$ where the wave velocity is decreased by a factor s_f. Report what happens with a Gaussian pulse, a "cosine hat" pulse, half a "cosine hat" pulse, and a plug pulse for resolutions $N_x = 40, 80, 160$, and $s_f = 2, 4$. Use $C = 1$ in the medium outside $[0.7, 0.9]$. Simulate until $T = 2$.
Filename: `pulse1D.py`.

Exercise 9: Compare discretizations of a Neumann condition

We have a 1D wave equation with variable wave velocity: $u_t = (qu_x)_x$. A Neumann condition u_x at $x = 0, L$ can be discretized as shown in (52) and (55).

The aim of this exercise is to examine the rate of the numerical error when using different ways of discretizing the Neumann condition. As test problem, $q = 1 + (x - L/2)^4$ can be used, with $f(x, t)$ adapted such that the solution has a simple form, say $u(x, t) = \cos(\pi x/L)\cos(\omega t)$ for some $\omega = \sqrt{q}\pi/L$.

a) Perform numerical experiments and find the convergence rate of the error using the approximation and (55).

b) Switch to $q(x) = \cos(\pi x/L)$, which is symmetric at $x = 0, L$, and check the convergence rate of the scheme (55). Now, $q_{i-1/2}$ is a 2nd-order approximation to q_i, $q_{i-1/2} = q_i + 0.25 q_i'' \Delta x^2 + \cdots$, because $q_i' = 0$ for $i = N_x$ (a similar argument can be applied to the case $i = 0$).

c) A third discretization can be based on a simple and convenient, but less accurate, one-sided difference: $u_i - u_{i-1} = 0$ at $i = N_x$ and $u_{i+1} - u_i = 0$ at $i = 0$. Derive the resulting scheme in detail and implement it. Run experiments to establish the rate of convergence.

d) A fourth technique is to view the scheme as

$$[D_t D_t u]_i^n = \frac{1}{\Delta x}\left([qD_x u]_{i+\frac{1}{2}}^n - [qD_x u]_{i-\frac{1}{2}}^n\right) + [f]_i^n,$$

and place the boundary at $x_{i+\frac{1}{2}}$, $i = N_x$, instead of exactly at the physical boundary. With this idea, we can just set $[qD_x u]_{i+\frac{1}{2}}^n = 0$. Derive the complete scheme using this technique. The implementation of the boundary condition at $L - \Delta x/2$ is $\mathcal{O}(\Delta x^2)$ accurate, but the interesting question is what impact the movement of the boundary has on the convergence rate (compute the errors as usual over the entire mesh).

Exercise 10: Verification by a cubic polynomial in space

The purpose of this exercise is to verify the implementation of the `solver` function in the program `wave1D_n0.py` by using an exact numerical solution for the wave equation $u_{tt} = c^2 u_{xx} + f$ with Neumann boundary conditions $u_x(0,t) = u_x(L,t) = 0$.

A similar verification is used in the file `wave1D_u0.py`, which solves the same PDE, but with Dirichlet boundary conditions $u(0,t) = u(L,t) = 0$. The idea of the verification test in function `test_quadratic` in `wave1D_u0.py` is to a solution that is a lower-order polynomial such that both the PDE problem, the boundary conditions, and all the discrete equations are exactly fulfilled. Then the `solver` function should reproduce this exact solution to machine precision. More precisely, we seek $u = X(x)T(t)$, with $T(t)$ as a linear function and $X(x)$ as a parabola that fulfills the boundary conditions. Inserting this u in the PDE determines f. It tuns out that u also fulfills the discrete equations, because the truncation error of the discretized PDE has derivatives in x and t of order four and higher. These derivatives all vanish for a quadratic $X(x)$ and linear $T(t)$.

It would be attractive to use a similar approach in the case of Neumann conditions. We set $u = X(x)T(t)$ and seek lower-order polynomials X and T. To force u_x to vanish at the boundary, we let X_x be a parabola. Then X is a cubic polynomial. The fourth-order derivative of a cubic polynomial vanishes, so $u = X(x)T(t)$ will fulfill the discretized PDE also in this case, if f is adjusted such that u fulfills the PDE.

However, the discrete boundary condition is not exactly fulfilled by this choice of u. The reason is that

$$[D_{2x}u]_i^n = u_x(x_i,t_n) + \frac{1}{6}u_{xxx}(x_i,t_n)\Delta x^2 + \mathcal{O}(\Delta x^4)\,. \tag{67}$$

At the boundary two boundary points, $X_x(x) = 0$ such that $u_x = 0$. However, u_{xxx} is a constant and not zero when $X(x)$ is a cubic polynomial. Therefore, our $u = X(x)T(t)$ fulfills

$$[D_{2x}u]_i^n = \frac{1}{6}u_{xxx}(x_i,t_n)\Delta x^2,$$

and not

$$[D_{2x}u]_i^n = 0, quadi = 0, N_x,$$

as it should. (Note that all the higher-order terms $\mathcal{O}(\Delta x^4)$ also have higher-order derivatives that vanish for a cubic polynomial.) So to summarize, the fundamental problem is that u as a product of a cubic polynomial and a linear or quadratic polynomial in time is not an exact solution of the discrete boundary conditions.

To make progress, we assume that $u = X(x)T(t)$, where T for simplicity is taken as a prescribed linear function $1 + \frac{1}{2}t$, and $X(x)$ is taken as an *unknown* cubic polynomial $\sum_{j=0}^{3} a_j x^j$. There are two different ways of determining the coefficients $a_0, \ldots, a_3$ such that both the discretized PDE and the discretized boundary conditions are fulfilled, under the constraint that we can specify a function $f(x,t)$ for the PDE to feed to the `solver` function in `wave1D_n0.py`. Both approaches are explained in the subexercises.

a) One can insert u in the discretized PDE and find the corresponding f. Then one can insert u in the discretized boundary conditions. This yields two equations for the four coefficients $a_0, \ldots, a_3$. To find the coefficients, one can set $a_0 = 0$ and $a_1 = 1$ for simplicity and then determine a_2 and a_3. This approach will make a_2 and a_3 depend on Δx and f will depend on both Δx and Δt.

Use `sympy` to perform analytical computations. A starting point is to define u as follows:

```
def test_cubic1():
    import sympy as sm
    x, t, c, L, dx, dt = sm.symbols('x t c L dx dt')
    i, n = sm.symbols('i n', integer=True)

    # Assume discrete solution is a polynomial of degree 3 in x
    T = lambda t: 1 + sm.Rational(1,2)*t  # Temporal term
    a = sm.symbols('a_0 a_1 a_2 a_3')
    X = lambda x: sum(a[q]*x**q for q in range(4))  # Spatial term
    u = lambda x, t: X(x)*T(t)
```

The symbolic expression for u is reached by calling `u(x,t)` with `x` and `t` as `sympy` symbols.

Define `DxDx(u, i, n)`, `DtDt(u, i, n)`, and `D2x(u, i, n)` as Python functions for returning the difference approximations $[D_xD_xu]_i^n$, $[D_tD_tu]_i^n$, and

$[D_{2x}u]^n_i$. The next step is to set up the residuals for the equations $[D_{2x}u]^n_0 = 0$ and $[D_{2x}u]^n_{N_x} = 0$, where $N_x = L/\Delta x$. Call the residuals `R_0` and `R_L`. Substitute a_0 and a_1 by 0 and 1, respectively, in `R_0`, `R_L`, and `a`:

```
R_0 = R_0.subs(a[0], 0).subs(a[1], 1)
R_L = R_L.subs(a[0], 0).subs(a[1], 1)
a = list(a)  # enable in-place assignment
a[0:2] = 0, 1
```

Determining a_2 and a_3 from the discretized boundary conditions is then about solving two equations with respect to a_2 and a_3, i.e., `a[2:]`:

```
s = sm.solve([R_0, R_L], a[2:])
# s is dictionary with the unknowns a[2] and a[3] as keys
a[2:] = s[a[2]], s[a[3]]
```

Now, `a` contains computed values and `u` will automatically use these new values since `X` accesses `a`.

Compute the source term f from the discretized PDE: $f^n_i = [D_tD_tu - c^2D_xD_xu]^n_i$. Turn u, the time derivative u_t (needed for the initial condition $V(x)$), and f into Python functions. Set numerical values for L, N_x, C, and c. Prescribe the time interval as $\Delta t = CL/(N_xc)$, which imply $\Delta x = c\Delta t/C = L/N_x$. Define new functions `I(x)`, `V(x)`, and `f(x,t)` as wrappers of the ones made above, where fixed values of L, c, Δx, and Δt are inserted, such that `I`, `V`, and `f` can be passed on to the `solver` function. Finally, call `solver` with a `user_action` function that compares the numerical solution to this exact solution u of the discrete PDE problem.

Hint. To turn a `sympy` expression `e`, depending on a series of symbols, say `x`, `t`, `dx`, `dt`, `L`, and `c`, into plain Python function `e_exact(x,t,L,dx,dt,c)`, one can write

```
e_exact = sm.lambdify([x,t,L,dx,dt,c], e, 'numpy')
```

The `'numpy'` argument is a good habit as the `e_exact` function will then work with array arguments if it contains mathematical functions (but here we only do plain arithmetics, which automatically work with arrays).

b) An alternative way of determining $a_0, \ldots, a_3$ is to reason as follows. We first construct $X(x)$ such that the boundary conditions are fulfilled: $X = x(L-x)$. However, to compensate for the fact that this choice of X does not fulfill the discrete boundary condition, we seek u such that

$$u_x = \frac{\partial}{\partial x}x(L-x)T(t) - \frac{1}{6}u_{xxx}\Delta x^2,$$

since this u will fit the discrete boundary condition. Assuming $u = T(t)\sum_{j=0}^3 a_jx^j$, we can use the above equation to determine the coefficients a_1, a_2, a_3. A value, e.g., 1 can be used for a_0. The following `sympy` code computes this u:

```
def test_cubic2():
    import sympy as sm
    x, t, c, L, dx = sm.symbols('x t c L dx')
    T = lambda t: 1 + sm.Rational(1,2)*t  # Temporal term
    # Set u as a 3rd-degree polynomial in space
    X = lambda x: sum(a[i]*x**i for i in range(4))
    a = sm.symbols('a_0 a_1 a_2 a_3')
    u = lambda x, t: X(x)*T(t)
    # Force discrete boundary condition to be zero by adding
    # a correction term the analytical suggestion x*(L-x)*T
    # u_x = x*(L-x)*T(t) - 1/6*u_xxx*dx**2
    R = sm.diff(u(x,t), x) - (
        x*(L-x) - sm.Rational(1,6)*sm.diff(u(x,t), x, x, x)*dx**2)
    # R is a polynomial: force all coefficients to vanish.
    # Turn R to Poly to extract coefficients:
    R = sm.poly(R, x)
    coeff = R.all_coeffs()
    s = sm.solve(coeff, a[1:])  # a[0] is not present in R
    # s is dictionary with a[i] as keys
    # Fix a[0] as 1
    s[a[0]] = 1
    X = lambda x: sm.simplify(sum(s[a[i]]*x**i for i in range(4)))
    u = lambda x, t: X(x)*T(t)
    print 'u:', u(x,t)
```

The next step is to find the source term `f_e` by inserting `u_e` in the PDE. Thereafter, turn `u`, `f`, and the time derivative of `u` into plain Python functions as in a), and then wrap these functions in new functions `I`, `V`, and `f`, with the right signature as required by the `solver` function. Set parameters as in a) and check that the solution is exact to machine precision at each time level using an appropriate `user_action` function.
Filename: `wave1D_n0_test_cubic.py`.

10 Analysis of the difference equations

10.1 Properties of the solution of the wave equation

The wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2}$$

has solutions of the form

$$u(x,t) = g_R(x-ct) + g_L(x+ct), \tag{68}$$

for any functions g_R and g_L sufficiently smooth to be differentiated twice. The result follows from inserting (68) in the wave equation. A function of the form $g_R(x-ct)$ represents a signal moving to the right in time with constant velocity c. This feature can be explained as follows. At time $t=0$ the signal looks like $g_R(x)$. Introducing a moving x axis with coordinates $\xi = x-ct$, we see the function $g_R(\xi)$ is "at rest" in the ξ coordinate system, and the shape is always the

same. Say the $g_R(\xi)$ function has a peak at $\xi = 0$. This peak is located at $x = ct$, which means that it moves with the velocity $dx/dt = c$ in the x coordinate system. Similarly, $g_L(x + ct)$ is a function initially with shape $g_L(x)$ that moves in the negative x direction with constant velocity c (introduce $\xi = x + ct$, look at the point $\xi = 0$, $x = -ct$, which has velocity $dx/dt = -c$).

With the particular initial conditions

$$u(x, 0) = I(x), \quad \frac{\partial}{\partial t} u(x, 0) = 0,$$

we get, with u as in (68),

$$g_R(x) + g_L(x) = I(x), \quad -cg_R'(x) + cg_L'(x) = 0,$$

which have the solution $g_R = g_L = I/2$, and consequently

$$u(x, t) = \frac{1}{2} I(x - ct) + \frac{1}{2} I(x + ct)\,. \tag{69}$$

The interpretation of (69) is that the initial shape of u is split into two parts, each with the same shape as I but half of the initial amplitude. One part is traveling to the left and the other one to the right.

The solution has two important physical features: constant amplitude of the left and right wave, and constant velocity of these two waves. It turns out that the numerical solution will also preserve the constant amplitude, but the velocity depends on the mesh parameters Δt and Δx.

The solution (69) will be influenced by boundary conditions when the parts $\frac{1}{2} I(x - ct)$ and $\frac{1}{2} I(x + ct)$ hit the boundaries and get, e.g., reflected back into the domain. However, when $I(x)$ is nonzero only in a small part in the middle of the spatial domain $[0, L]$, which means that the boundaries are placed far away from the initial disturbance of u, the solution (69) is very clearly observed in a simulation.

A useful representation of solutions of wave equations is a linear combination of sine and/or cosine waves. Such a sum of waves is a solution if the governing PDE is linear and each sine or cosine wave fulfills the equation. To ease analytical calculations by hand we shall work with complex exponential functions instead of real-valued sine or cosine functions. The real part of complex expressions will typically be taken as the physical relevant quantity (whenever a physical relevant quantity is strictly needed). The idea now is to build $I(x)$ of complex wave components e^{ikx}:

$$I(x) \approx \sum_{k \in K} b_k e^{ikx}\,. \tag{70}$$

Here, k is the frequency of a component, K is some set of all the discrete k values needed to approximate $I(x)$ well, and b_k are constants that must be determined. We will very seldom need to compute the b_k coefficients: most of the insight we look for and the understanding of the numerical methods we want to establish, come from investigating how the PDE and the scheme treat a single component e^{ikx} wave.

Letting the number of k values in K tend to infinity makes the sum (70) converge to $I(x)$, and this sum is known as a *Fourier series* representation of $I(x)$. Looking at (69), we see that the solution $u(x,t)$, when $I(x)$ is represented as in (70), is also built of basic complex exponential wave components of the form $e^{ik(x\pm ct)}$ according to

$$u(x,t) = \frac{1}{2}\sum_{k\in K} b_k e^{ik(x-ct)} + \frac{1}{2}\sum_{k\in K} b_k e^{ik(x+ct)}\,. \tag{71}$$

It is common to introduce the frequency in time $\omega = kc$ and assume that $u(x,t)$ is a sum of basic wave components written as $e^{ikx-\omega t}$. (Observe that inserting such a wave component in the governing PDE reveals that $\omega^2 = k^2c^2$, or $\omega \pm kc$, reflecting the two solutions: one $(+kc)$ traveling to the right and the other $(-kc)$ traveling to the left.)

10.2 More precise definition of Fourier representations

The quick intuitive introduction above to representing a function by a sum of sine and cosine waves suffices as background for the forthcoming material on analyzing a single wave component. However, to understand all details of how different wave components sum up to the analytical and numerical solution, a more precise mathematical treatment is helpful and therefore summarized below.

It is well known that periodic functions can be represented by Fourier series. A generalization of the Fourier series idea to non-periodic functions defined on the real line is the *Fourier transform*:

$$I(x) = \int_{-\infty}^{\infty} A(k)e^{ikx}dk, \tag{72}$$

$$A(k) = \int_{-\infty}^{\infty} I(x)e^{-ikx}dx\,. \tag{73}$$

The function $A(k)$ reflects the weight of each wave component e^{ikx} in an infinite sum of such wave components. That is, $A(k)$ reflects the frequency content in the function $I(x)$. Fourier transforms are particularly fundamental for analyzing and understanding time-varying signals.

The solution of the linear 1D wave PDE can be expressed as

$$u(x,t) = \int_{-\infty}^{\infty} A(k)e^{i(kx-\omega(k)t)}dx\,.$$

In a finite difference method, we represent u by a mesh function u_q^n, where n counts temporal mesh points and q counts the spatial ones (the usual counter for spatial points, i, is here already used as imaginary unit). Similarly, $I(x)$ is approximated by the mesh function I_q, $q = 0,\ldots,N_x$. On a mesh, it does not make sense to work with wave components e^{ikx} for very large k, because the shortest possible sine or cosine wave that can be represented on a mesh

with spacing Δx is the wave with wavelength $2\Delta x$ (the sine/cosine signal jumps up and down between each mesh point). The corresponding k value is $k = 2\pi/(2\Delta x) = \pi/\Delta x$, known as the *Nyquist frequency*. Within the range of relevant frequencies $(0, \pi/\Delta x]$ one defines the discrete Fourier transform, using $N_x + 1$ discrete frequencies:

$$I_q = \frac{1}{N_x+1}\sum_{k=0}^{N_x} A_k e^{i2\pi kj/(N_x+1)}, \quad i = 0, \ldots, N_x, \tag{74}$$

$$A_k = \sum_{q=0}^{N_x} I_q e^{-i2\pi kq/(N_x+1)}, \quad k = 0, \ldots, N_x + 1\,. \tag{75}$$

The A_k values is the discrete Fourier transform of the I_q values, and the latter are the inverse discrete Fourier transform of the A_k values.

The discrete Fourier transform is efficiently computed by the *Fast Fourier transform* algorithm. For a real function $I(x)$ the relevant Python code for computing and plotting the discrete Fourier transform appears in the example below.

```
import numpy as np
from numpy import sin

def I(x):
    return sin(2*pi*x) + 0.5*sin(4*pi*x) + 0.1*sin(6*pi*x)

# Mesh
L = 10; Nx = 100
x = np.linspace(0, L, Nx+1)
dx = L/float(Nx)

# Discrete Fourier transform
A = np.fft.rfft(I(x))
A_amplitude = np.abs(A)

# Compute the corresponding frequencies
freqs = np.linspace(0, pi/dx, A_amplitude.size)

import matplotlib.pyplot as plt
plt.plot(freqs, A_amplitude)
plt.show()
```

10.3 Stability

The scheme

$$[D_tD_tu = c^2D_xD_xu]^n_q \tag{76}$$

for the wave equation $u_t = c^2u_{xx}$ allows basic wave components

$$u^n_q = e^{i(kx_q - \tilde{\omega}t_n)}$$

as solution, but it turns out that the frequency in time, $\tilde{\omega}$, is not equal to the exact $\omega = kc$. The idea now is to study how the scheme treats an arbitrary wave component with a given k. We ask two key questions:

- How accurate is $\tilde{\omega}$ compared to ω?
- Does the amplitude of such a wave component preserve its (unit) amplitude, as it should, or does it get amplified or damped in time (due to a complex $\tilde{\omega}$)?

The following analysis will answer these questions. Note the need for using q as counter for the mesh point in x direction since i is already used as the imaginary unit (in this analysis).

Preliminary results. A key result needed in the investigations is the finite difference approximation of a second-order derivative acting on a complex wave component:

$$[D_tD_te^{i\omega t}]^n = -\frac{4}{\Delta t^2}\sin^2\left(\frac{\omega\Delta t}{2}\right)e^{i\omega n\Delta t}\,.$$

By just changing symbols ($\omega \rightarrow k$, $t \rightarrow x$, $n \rightarrow q$) it follows that

$$[D_xD_xe^{ikx}]_q = -\frac{4}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right)e^{ikq\Delta x}\,.$$

Numerical wave propagation. Inserting a basic wave component $u_q^n = e^{i(kx_q-\tilde{\omega}t_n)}$ in (76) results in the need to evaluate two expressions:

$$\begin{aligned}
[D_tD_te^{ikx}e^{-i\tilde{\omega}t}]_q^n &= [D_tD_te^{-i\tilde{\omega}t}]^n e^{ikq\Delta x}\\
&= -\frac{4}{\Delta t^2}\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right)e^{-i\tilde{\omega}n\Delta t}e^{ikq\Delta x} \qquad (77)\\
[D_xD_xe^{ikx}e^{-i\tilde{\omega}t}]_q^n &= [D_xD_xe^{ikx}]_q e^{-i\tilde{\omega}n\Delta t}\\
&= -\frac{4}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right)e^{ikq\Delta x}e^{-i\tilde{\omega}n\Delta t}\,. \qquad (78)
\end{aligned}$$

Then the complete scheme,

$$[D_tD_te^{ikx}e^{-i\tilde{\omega}t} = c^2D_xD_xe^{ikx}e^{-i\tilde{\omega}t}]_q^n$$

leads to the following equation for the unknown numerical frequency $\tilde{\omega}$ (after dividing by $-e^{ikx}e^{-i\tilde{\omega}t}$):

$$\frac{4}{\Delta t^2}\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = c^2\frac{4}{\Delta x^2}\sin^2\left(\frac{k\Delta x}{2}\right),$$

or

$$\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C^2\sin^2\left(\frac{k\Delta x}{2}\right), \tag{79}$$

where

$$C = \frac{c\Delta t}{\Delta x} \tag{80}$$

is the Courant number. Taking the square root of (79) yields

$$\sin\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C\sin\left(\frac{k\Delta x}{2}\right), \tag{81}$$

Since the exact ω is real it is reasonable to look for a real solution $\tilde{\omega}$ of (81). The right-hand side of (81) must then be in $[-1, 1]$ because the sine function on the left-hand side has values in $[-1, 1]$ for real $\tilde{\omega}$. The sine function on the right-hand side can attain the value 1 when

$$\frac{k\Delta x}{2} = m\frac{\pi}{2}, \quad m \in \mathbb{Z}\,.$$

With $m = 1$ we have $k\Delta x = \pi$, which means that the wavelength $\lambda = 2\pi/k$ becomes $2\Delta x$. This is the absolutely shortest wavelength that can be represented on the mesh: the wave jumps up and down between each mesh point. Larger values of $|m|$ are irrelevant since these correspond to k values whose waves are too short to be represented on a mesh with spacing Δx. For the shortest possible wave in the mesh, $\sin(k\Delta x/2) = 1$, and we must require

$$C \leq 1\,. \tag{82}$$

Consider a right-hand side in (81) of magnitude larger than unity. The solution $\tilde{\omega}$ of (81) must then be a complex number $\tilde{\omega} = \tilde{\omega}_r + i\tilde{\omega}_i$ because the sine function is larger than unity for a complex argument. One can show that for any ω_i there will also be a corresponding solution with $-\omega_i$. The component with $\omega_i > 0$ gives an amplification factor $e^{\omega_i t}$ that grows exponentially in time. We cannot allow this and must therefore require $C \leq 1$ as a *stability criterion*.

Remark.

For smoother wave components with longer wave lengths per length Δx, (82) can in theory be relaxed. However, small round-off errors are always present in a numerical solution and these vary arbitrarily from mesh point to mesh point and can be viewed as unavoidable noise with wavelength $2\Delta x$. As explained, $C > 1$ will for this very small noise lead to exponential growth of the shortest possible wave component in the mesh. This noise will therefore grow with time and destroy the whole solution.

10.4 Numerical dispersion relation

Equation (81) can be solved with respect to $\tilde{\omega}$:

$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(C\sin\left(\frac{k\Delta x}{2}\right)\right) . \tag{83}$$

The relation between the numerical frequency $\tilde{\omega}$ and the other parameters k, c, Δx, and Δt is called a *numerical dispersion relation*. Correspondingly, $\omega = kc$ is the *analytical dispersion relation*.

The special case $C = 1$ deserves attention since then the right-hand side of (83) reduces to

$$\frac{2}{\Delta t}\frac{k\Delta x}{2} = \frac{1}{\Delta t}\frac{\omega\Delta x}{c} = \frac{\omega}{C} = \omega .$$

That is, $\tilde{\omega} = \omega$ and the numerical solution is exact at all mesh points regardless of Δx and Δt! This implies that the numerical solution method is also an analytical solution method, at least for computing u at discrete points (the numerical method says nothing about the variation of u *between* the mesh points, and employing the common linear interpolation for extending the discrete solution gives a curve that deviates from the exact one).

For a closer examination of the error in the numerical dispersion relation when $C < 1$, we can study $\tilde{\omega} - \omega$, $\tilde{\omega}/\omega$, or the similar error measures in wave velocity: $\tilde{c} - c$ and $\tilde{c}/c$, where $c = \omega/k$ and $\tilde{c} = \tilde{\omega}/k$. It appears that the most convenient expression to work with is $\tilde{c}/c$:

$$\frac{\tilde{c}}{c} = \frac{1}{Cp}\sin^{-1}\left(C\sin p\right) ,$$

with $p = k\Delta x/2$ as a non-dimensional measure of the spatial frequency. In essence, p tells how many spatial mesh points we have per wave length in space of the wave component with frequency k (the wave length is $2\pi/k$). That is, p reflects how well the spatial variation of the wave component is resolved in the mesh. Wave components with wave length less than $2\Delta x$ ($2\pi/k < 2\Delta x$) are not visible in the mesh, so it does not make sense to have $p > \pi/2$.

We may introduce the function $r(C, p) = \tilde{c}/c$ for further investigation of numerical errors in the wave velocity:

$$r(C, p) = \frac{1}{Cp}\sin^{-1}\left(C\sin p\right) , \quad C \in (0, 1],\ p \in (0, \pi/2] . \tag{84}$$

This function is very well suited for plotting since it combines several parameters in the problem into a dependence on two non-dimensional numbers, C and p.

Defining

```
def r(C, p):
    return 2/(C*p)*asin(C*sin(p))
```

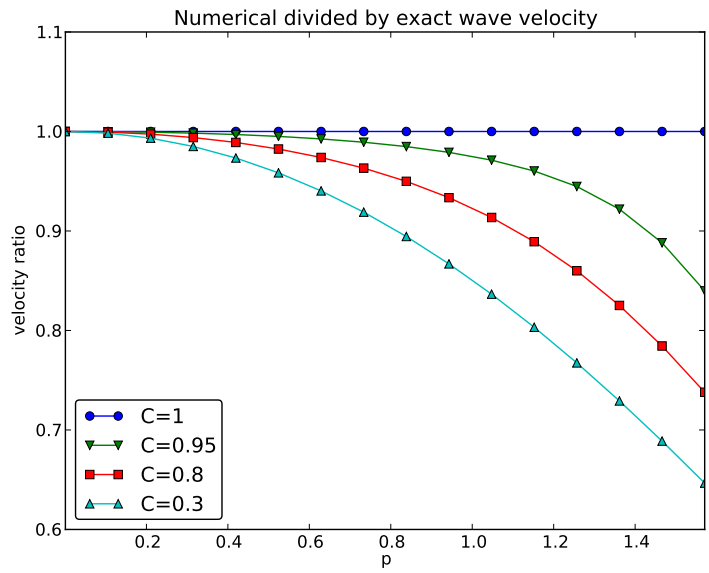

Figure 6: The fractional error in the wave velocity for different Courant numbers.

we can plot $r(C, p)$ as a function of p for various values of C, see Figure 6. Note that the shortest waves have the most erroneous velocity, and that short waves move more slowly than they should.

With `sympy` we can also easily make a Taylor series expansion in the discretization parameter p:

```
>>> C, p = symbols('C p')
>>> # Compute the 7 first terms around p=0 with no O() term
>>> rs = r(C, p).series(p, 0, 7).removeO()
>>> rs
p**6*(5*C**6/112 - C**4/16 + 13*C**2/720 - 1/5040) +
p**4*(3*C**4/40 - C**2/12 + 1/120) +
p**2*(C**2/6 - 1/6) + 1
>>> rs_error_leading_order = (rs - 1).extract_leading_order(p)
>>> rs_error_leading_order
p**2*(C**2/6 - 1/6)
>>> # Turn the series expansion into a Python function
>>> rs_pyfunc = lambdify([C, p], rs, modules='numpy')
>>> # Check: rs_pyfunc is exact (=1) for C=1
>>> rs_pyfunc(1, 0.1)
1.0
```

Without the `.removeO()` call the series get an `O(x**7)` term that makes it impossible to convert the series to a Python function (for, e.g., plotting).

From the `rs_error_leading_order` expression above we see that the leading order term in the error of this series expansion is

$$\frac{1}{6}\left(\frac{k\Delta x}{2}\right)^2 (C^2-1) = \frac{k^2}{24}\left(c^2\Delta t^2 - \Delta x^2\right), \tag{85}$$

pointing to an error $\mathcal{O}(\Delta t^2, \Delta x^2)$, which is compatible with the errors in the difference approximations (D_tD_t and D_xD_x).

Here is an alternative way of performing a series expansion: we use the `lseries` method, which returns an iterator over all the terms in the expansion, and ask for the 4 first terms (via `itertools.islice`, which can slice an iterator). Collecting the terms in a list makes it possible to factor each term individually. Summing up the terms results in a nicer expression:

```
>>> import itertools
>>> rs = [t for t in itertools.islice(r(C, p).lseries(p), 4)]
>>> rs
[1, C**2*p**2/6 - p**2/6,
 3*C**4*p**4/40 - C**2*p**4/12 + p**4/120,
 5*C**6*p**6/112 - C**4*p**6/16 + 13*C**2*p**6/720 - p**6/5040]
>>> rs = [factor(t) for t in rs]
>>> rs
[1, p**2*(C - 1)*(C + 1)/6,
 p**4*(C - 1)*(C + 1)*(3*C - 1)*(3*C + 1)/120,
 p**6*(C - 1)*(C + 1)*(225*C**4 - 90*C**2 + 1)/5040]
>>> rs = sum(rs)  # Python's sum function
>>> rs
p**6*(C - 1)*(C + 1)*(225*C**4 - 90*C**2 + 1)/5040 +
p**4*(C - 1)*(C + 1)*(3*C - 1)*(3*C + 1)/120 +
p**2*(C - 1)*(C + 1)/6 + 1
```

We see from the last expression that $C = 1$ makes all the terms in `rs` vanish. Since we already know that the numerical solution is exact for $C = 1$, the remaining terms in the Taylor series expansion will also contain factors of $C - 1$ and cancel for $C = 1$.

10.5 Extending the analysis to 2D and 3D

The typical analytical solution of a 2D wave equation

$$u_{tt} = c^2(u_{xx} + u_{yy}),$$

is a wave traveling in the direction of $\boldsymbol{k} = k_x\boldsymbol{i} + k_y\boldsymbol{j}$, where $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in the x and y directions, respectively. Such a wave can be expressed by

$$u(x, y, t) = g(k_x x + k_y y - kct)$$

for some twice differentiable function g, or with $\omega = kc$, $k = |\boldsymbol{k}|$:

$$u(x, y, t) = g(k_x x + k_y y - \omega t)\,.$$

We can in particular build a solution by adding complex Fourier components of the form

$$\exp\left(i(k_x x + k_y y - \omega t)\right).$$

A discrete 2D wave equation can be written as

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u)]^n_{q,r}\,. \tag{86}$$

This equation admits a Fourier component

$$u^n_{q,r} = \exp\left(i(k_x q\Delta x + k_y r\Delta y - \tilde{\omega} n\Delta t)\right), \tag{87}$$

as solution. Letting the operators $D_t D_t$, $D_x D_x$, and $D_y D_y$ act on $u^n_{q,r}$ from (87) transforms (86) to

$$\frac{4}{\Delta t^2}\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = c^2\frac{4}{\Delta x^2}\sin^2\left(\frac{k_x\Delta x}{2}\right) + c^2\frac{4}{\Delta y^2}\sin^2\left(\frac{k_y\Delta y}{2}\right). \tag{88}$$

or

$$\sin^2\left(\frac{\tilde{\omega}\Delta t}{2}\right) = C_x^2\sin^2 p_x + C_y^2\sin^2 p_y, \tag{89}$$

where we have eliminated the factor 4 and introduced the symbols

$$C_x = \frac{c^2\Delta t^2}{\Delta x^2},\quad C_y = \frac{c^2\Delta t^2}{\Delta y^2},\quad p_x = \frac{k_x\Delta x}{2},\quad p_y = \frac{k_y\Delta y}{2}\,.$$

For a real-valued $\tilde{\omega}$ the right-hand side must be less than or equal to unity in absolute value, requiring in general that

$$C_x^2 + C_y^2 \leq 1\,. \tag{90}$$

This gives the stability criterion, more commonly expressed directly in an inequality for the time step:

$$\Delta t \leq \frac{1}{c}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-1/2} \tag{91}$$

A similar, straightforward analysis for the 3D case leads to

$$\Delta t \leq \frac{1}{c}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)^{-1/2} \tag{92}$$

In the case of a variable coefficient $c^2 = c^2(\boldsymbol{x})$, we must use the worst-case value

$$\bar{c} = \sqrt{\max_{\boldsymbol{x}\in\Omega} c^2(\boldsymbol{x})} \tag{93}$$

in the stability criteria. Often, especially in the variable wave velocity case, it is wise to introduce a safety factor $\beta \in (0, 1]$ too:

$$\Delta t \leq \beta\frac{1}{\bar{c}}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)^{-1/2} \tag{94}$$

The exact numerical dispersion relations in 2D and 3D becomes, for constant c,

$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\left(C_x^2\sin^2 p_x + C_y^2\sin_y^p\right)^{\frac{1}{2}}\right), \tag{95}$$

$$\tilde{\omega} = \frac{2}{\Delta t}\sin^{-1}\left(\left(C_x^2\sin^2 p_x + C_y^2\sin_y^p + C_z^2\sin_z^p\right)^{\frac{1}{2}}\right). \tag{96}$$

We can visualize the numerical dispersion error in 2D much like we did in 1D. To this end, we need to reduce the number of parameters in $\tilde{\omega}$. The direction of the wave is parameterized by the polar angle θ, which means that

$$k_x = k\sin\theta, \quad k_y = k\cos\theta\,.$$

A simplification is to set $\Delta x = \Delta y = h$. Then $C_x = C_y = c\Delta t/h$, which we call C. Also,

$$p_x = \frac{1}{2}kh\cos\theta, \quad p_y = \frac{1}{2}kh\sin\theta\,.$$

The numerical frequency $\tilde{\omega}$ is now a function of three parameters:

- C reflecting the number cells a wave is displaced during a time step
- kh reflecting the number of cells per wave length in space
- θ expressing the direction of the wave

We want to visualize the error in the numerical frequency. To avoid having Δt as a free parameter in $\tilde{\omega}$, we work with $\tilde{c}/c$, because the fraction $2/\Delta t$ is then rewritten as

$$\frac{2}{kc\Delta t} = \frac{2}{2kc\Delta t h/h} = \frac{1}{Ckh},$$

and

$$\frac{\tilde{c}}{c} = \frac{1}{Ckh}\sin^{-1}\left(C\left(\sin^2(\frac{1}{2}kh\cos\theta) + \sin^2(\frac{1}{2}kh\sin\theta)\right)^{\frac{1}{2}}\right).$$

We want to visualize this quantity as a function of kh and θ for some values of $C \leq 1$. It is instructive to make color contour plots of $1-\tilde{c}/c$ in *polar coordinates* with θ as the angular coordinate and kh as the radial coordinate.

The stability criterion (90) becomes $C \leq C_{\max} = 1/\sqrt{2}$ in the present 2D case with the C defined above. Let us plot $1-\tilde{c}/c$ in polar coordinates for $C_{\max}, 0.9C_{\max}, 0.5C_{\max}, 0.2C_{\max}$. The program below does the somewhat tricky work in Matplotlib, and the result appears in Figure 7. From the figure we clearly see that the maximum C value gives the best results, and that waves whose propagation direction makes an angle of 45 degrees with an axis are the most accurate.

```
def dispersion_relation_2D(kh, theta, C):
    arg = C*sqrt(sin(0.5*kh*cos(theta))**2 +
                 sin(0.5*kh*sin(theta))**2)
    c_frac = 2./(C*kh)*arcsin(arg)

    return c_frac

from numpy import exp, sin, cos, linspace, \
     pi, meshgrid, arcsin, sqrt
r = kh = linspace(0.001, pi, 101)
theta = linspace(0, 2*pi, 51)
r, theta = meshgrid(r, theta)

# Make 2x2 filled contour plots for 4 values of C
import matplotlib.pyplot as plt
C_max = 1/sqrt(2)
C = [[C_max, 0.9*C_max], [0.5*C_max, 0.2*C_max]]
fix, axes = plt.subplots(2, 2, subplot_kw=dict(polar=True))
for row in range(2):
    for column in range(2):
        error = 1 - dispersion_relation_2D(
            kh, theta, C[row][column])
        print error.min(), error.max()
        cax = axes[row][column].contourf(
            theta, r, error, 50, vmin=0, vmax=0.36)
        axes[row][column].set_xticks([])
        axes[row][column].set_yticks([])

# Add colorbar to the last plot
cbar = plt.colorbar(cax)
cbar.ax.set_ylabel('error in wave velocity')
plt.savefig('disprel2D.png')
plt.savefig('disprel2D.pdf')
plt.show()
```

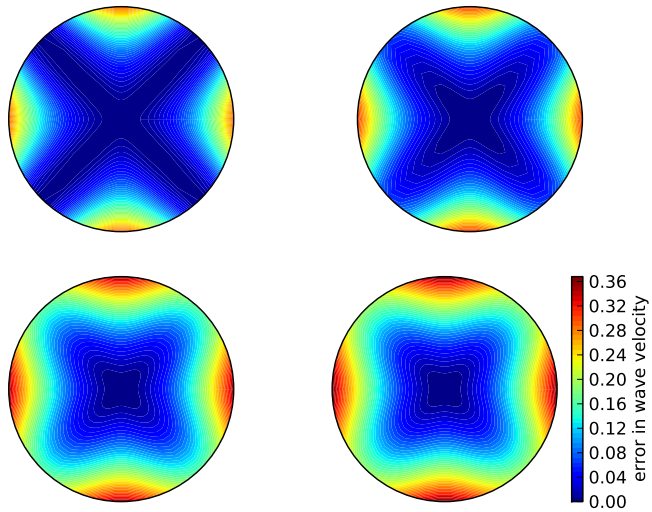

Figure 7: Error in numerical dispersion in 2D.

11 Finite difference methods for 2D and 3D wave equations

A natural next step is to consider extensions of the methods for various variants of the one-dimensional wave equation to two-dimensional (2D) and three-dimensional (3D) versions of the wave equation.

11.1 Multi-dimensional wave equations

The general wave equation in d space dimensions, with constant wave velocity c, can be written in the compact form

$$\frac{\partial^2 u}{\partial t^2} = c^2\nabla^2 u \text{ for } \boldsymbol{x} \in \Omega \subset \mathbb{R}^d,\ t \in (0, T]\,. \tag{97}$$

In a 2D problem ($d = 2$),

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2},$$

while in three space dimensions ($d = 3$),

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\,.$$

Many applications involve variable coefficients, and the general wave equation in d dimensions is in this case written as

$$\varrho\frac{\partial^2 u}{\partial t^2} = \nabla\cdot(q\nabla u) + f \text{ for } \boldsymbol{x}\in\Omega\subset\mathbb{R}^d,\ t\in(0,T], \tag{98}$$

which in 2D becomes

$$\varrho(x,y)\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(q(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) + f(x,y,t)\,. \tag{99}$$

To save some writing and space we may use the index notation, where subscript t, x, y, or z means differentiation with respect to that coordinate. For example,

$$\frac{\partial^2 u}{\partial t^2} = u_{tt},$$
$$\frac{\partial}{\partial y}\left(q(x,y)\frac{\partial u}{\partial y}\right) = (qu_y)_y\,.$$

The 3D versions of the two model PDEs, with and without variable coefficients, can with now with the aid of the index notation for differentiation be stated as

$$u_{tt} = c^2(u_{xx} + u_{yy} + u_{zz}) + f, \tag{100}$$
$$\varrho u_{tt} = (qu_x)_x + (qu_z)_z + (qu_z)_z + f\,. \tag{101}$$

At *each point* of the boundary $\partial\Omega$ of Ω we need *one* boundary condition involving the unknown u. The boundary conditions are of three principal types:

1. u is prescribed ($u = 0$ or a known time variation for an incoming wave),
2. $\partial u/\partial n = \boldsymbol{n}\cdot\nabla u$ prescribed (zero for reflecting boundaries),
3. an open boundary condition (also called radiation condition) is specified to let waves travel undisturbed out of the domain, see Exercise **??** for details.

All the listed wave equations with *second-order* derivatives in time need *two* initial conditions:

1. $u = I$,
2. $u_t = V$.

11.2 Mesh

We introduce a mesh in time and in space. The mesh in time consists of time points

$$t_0 = 0 < t_1 < \cdots < t_{N_t},$$

often with a constant spacing $\Delta t = t_{n+1} - t_n$, $n \in \mathcal{I}_t^-$.

Finite difference methods are easy to implement on simple rectangle- or box-shaped domains. More complicated shapes of the domain require substantially more advanced techniques and implementational efforts. On a rectangle- or box-shaped domain mesh points are introduced separately in the various space directions:

$$\begin{aligned}
&x_0 < x_1 < \cdots < x_{N_x} \text{ in } x \text{ direction},\\
&y_0 < y_1 < \cdots < y_{N_y} \text{ in } y \text{ direction},\\
&z_0 < z_1 < \cdots < z_{N_z} \text{ in } z \text{ direction}.
\end{aligned}$$

We can write a general mesh point as (x_i, y_j, z_k, t_n), with $i \in \mathcal{I}_x$, $j \in \mathcal{I}_y$, $k \in \mathcal{I}_z$, and $n \in \mathcal{I}_t$.

It is a very common choice to use constant mesh spacings: $\Delta x = x_{i+1} - x_i$, $i \in \mathcal{I}_x^-$, $\Delta y = y_{j+1} - y_j$, $j \in \mathcal{I}_y^-$, and $\Delta z = z_{k+1} - z_k$, $k \in \mathcal{I}_z^-$. With equal mesh spacings one often introduces $h = \Delta x = \Delta y = \Delta z$.

The unknown u at mesh point (x_i, y_j, z_k, t_n) is denoted by $u^n_{i,j,k}$. In 2D problems we just skip the z coordinate (by assuming no variation in that direction: $\partial/\partial z = 0$) and write $u^n_{i,j}$.

11.3 Discretization

Two- and three-dimensional wave equations are easily discretized by assembling building blocks for discretization of 1D wave equations, because the multi-dimensional versions just contain terms of the same type that occurs in 1D.

Discretizing the PDEs. Equation (100) can be discretized as

$$[D_tD_tu = c^2(D_xD_xu + D_yD_yu + D_zD_zu) + f]^n_{i,j,k}\,. \tag{102}$$

A 2D version might be instructive to write out in detail:

$$[D_tD_tu = c^2(D_xD_xu + D_yD_yu) + f]^n_{i,j,k},$$

which becomes

$$\frac{u^{n+1}_{i,j} - 2u^n_{i,j} + u^{n-1}_{i,j}}{\Delta t^2} = c^2\frac{u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}}{\Delta x^2} + c^2\frac{u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}}{\Delta y^2} + f^n_{i,j},$$

Assuming as usual that all values at the time levels n and $n-1$ are known, we can solve for the only unknown $u^{n+1}_{i,j}$. The result can be compactly written as

$$u^{n+1}_{i,j} = 2u^n_{i,j} + u^{n-1}_{i,j} + c^2\Delta t^2[D_xD_xu + D_yD_yu]^n_{i,j}\,. \tag{103}$$

As in the 1D case, we need to develop a special formula for $u^1_{i,j}$ where we combine the general scheme for $u^{n+1}_{i,j}$, when $n=0$, with the discretization of the initial condition:

$$[D_{2t}u = V]^0_{i,j} \quad \Rightarrow \quad u^{-1}_{i,j} = u^1_{i,j} - 2\Delta t V_{i,j}\,.$$

The result becomes, in compact form,

$$u^{n+1}_{i,j} = u^n_{i,j} - 2\Delta V_{i,j} + \frac{1}{2}c^2\Delta t^2[D_xD_xu + D_yD_yu]^n_{i,j}\,. \tag{104}$$

The PDE (101) with variable coefficients is discretized term by term using the corresponding elements from the 1D case:

$$[\varrho D_tD_tu = (D_x\overline{q}^xD_xu + D_y\overline{q}^yD_yu + D_z\overline{q}^zD_zu) + f]^n_{i,j,k}\,. \tag{105}$$

When written out and solved for the unknown $u^{n+1}_{i,j,k}$, one gets the scheme

$$\begin{aligned}
u^{n+1}_{i,j,k} &= -u^{n-1}_{i,j,k} + 2u^n_{i,j,k} + \\
&= \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta x^2}(\frac{1}{2}(q_{i,j,k}+q_{i+1,j,k})(u^n_{i+1,j,k}-u^n_{i,j,k}) - \\
&\qquad \frac{1}{2}(q_{i-1,j,k}+q_{i,j,k})(u^n_{i,j,k}-u^n_{i-1,j,k})) + \\
&= \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta x^2}(\frac{1}{2}(q_{i,j,k}+q_{i,j+1,k})(u^n_{i,j+1,k}-u^n_{i,j,k}) - \\
&\qquad \frac{1}{2}(q_{i,j-1,k}+q_{i,j,k})(u^n_{i,j,k}-u^n_{i,j-1,k})) + \\
&= \frac{1}{\varrho_{i,j,k}}\frac{1}{\Delta x^2}(\frac{1}{2}(q_{i,j,k}+q_{i,j,k+1})(u^n_{i,j,k+1}-u^n_{i,j,k}) - \\
&\qquad \frac{1}{2}(q_{i,j,k-1}+q_{i,j,k})(u^n_{i,j,k}-u^n_{i,j,k-1})) + \\
&+ \qquad \Delta t^2 f^n_{i,j,k}\,.
\end{aligned}$$

Also here we need to develop a special formula for $u^1_{i,j,k}$ by combining the scheme for $n=0$ with the discrete initial condition, which is just a matter of inserting $u^{-1}_{i,j,k} = u^1_{i,j,k} - 2\Delta t V_{i,j,k}$ in the scheme and solving for $u^1_{i,j,k}$.

Handling boundary conditions where is u known. The schemes listed above are valid for the internal points in the mesh. After updating these, we need to visit all the mesh points at the boundaries and set the prescribed u value.

Discretizing the Neumann condition. The condition $\partial u/\partial n = 0$ was implemented in 1D by discretizing it with a $D_{2x}u$ centered difference, and thereafter eliminating the fictitious u point outside the mesh by using the general scheme at the boundary point. Alternatively, one can introduce ghost cells and update a ghost value to for use in the Neumann condition. Exactly the same ideas are reused in multi dimensions.

Consider $\partial u/\partial n = 0$ at a boundary $y = 0$. The normal direction is then in $-y$ direction, so

$$\frac{\partial u}{\partial n} = -\frac{\partial u}{\partial y},$$

and we set

$$[-D_{2y}u = 0]^n_{i,0} \quad \Rightarrow \quad \frac{u^n_{i,1} - u^n_{i,-1}}{2\Delta y} = 0\,.$$

From this it follows that $u^n_{i,-1} = u^n_{i,1}$. The discretized PDE at the boundary point $(i, 0)$ reads

$$\frac{u^{n+1}_{i,0} - 2u^n_{i,0} + u^{n-1}_{i,0}}{\Delta t^2} = c^2\frac{u^n_{i+1,0} - 2u^n_{i,0} + u^n_{i-1,0}}{\Delta x^2} + c^2\frac{u^n_{i,1} - 2u^n_{i,0} + u^n_{i,-1}}{\Delta y^2} + f^n_{i,j},$$

We can then just insert $u^1_{i,1}$ for $u^n_{i,-1}$ in this equation and then solve for the boundary value $u^{n+1}_{i,0}$ as done in 1D.

From these calculations, we see a pattern: the general scheme applies at the boundary $j = 0$ too if we just replace $j - 1$ by $j + 1$. Such a pattern is particularly useful for implementations. The details follow from the explained 1D case in Section 6.3.

The alternative approach to eliminating fictitious values outside the mesh is to have $u^n_{i,-1}$ available as a ghost value. The mesh is extended with one extra line (2D) or plane (3D) of ghost cells at a Neumann boundary. In the present example it means that we need a line ghost cells below the y axis. The ghost values must be updated according to $u^{n+1}_{i,-1} = u^{n+1}_{i,1}$.

12 Implementation

We shall now describe in detail various Python implementations for solving a standard 2D, linear wave equation with constant wave velocity and $u = 0$ on the boundary. The wave equation is to be solved in the space-time domain $\Omega \times (0, T]$, where $\Omega = (0, L_x) \times (0, L_y)$ is a rectangular spatial domain. More precisely, the complete initial-boundary value problem is defined by

$$u_t = c^2(u_{xx} + u_{yy}) + f(x, y, t), \qquad (x, y) \in \Omega,\ t \in (0, T], \tag{106}$$

$$u(x, y, 0) = I(x, y), \qquad (x, y) \in \Omega, \tag{107}$$

$$u_t(x, y, 0) = V(x, y), \qquad (x, y) \in \Omega, \tag{108}$$

$$u = 0, \qquad (x, y) \in \partial\Omega,\ t \in (0, T], \tag{109}$$

where $\partial\Omega$ is the boundary of Ω, in this case the four sides of the rectangle $[0, L_x] \times [0, L_y]$: $x = 0$, $x = L_x$, $y = 0$, and $y = L_y$.

The PDE is discretized as

$$[D_tD_tu = c^2(D_xD_xu + D_yD_yu) + f]^n_{i,j},$$

which leads to an explicit updating formula to be implemented in a program:

$$\begin{aligned} u^{n+1} = &-u^{n-1}_{i,j} + 2u^n_{i,j} + \\ &C_x^2(u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}) + C_y^2(u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}) + \Delta t^2 f^n_{i,j}, \end{aligned} \tag{110}$$

for all interior mesh points $i \in \mathcal{I}^i_x$ and $j \in \mathcal{I}^i_y$, and for $n \in \mathcal{I}^+_t$. The constants C_x and C_y are defined as

$$C_x = c\frac{\Delta t}{\Delta x}, \quad C_x = c\frac{\Delta t}{\Delta y}\,.$$

At the boundary we simply set $u^{n+1}_{i,j} = 0$ for $i = 0$, $j = 0, \ldots, N_y$; $i = N_x$, $j = 0, \ldots, N_y$; $j = 0$, $i = 0, \ldots, N_x$; and $j = N_y$, $i = 0, \ldots, N_x$. For the first step, $n = 0$, (111) is combined with the discretization of the initial condition $u_t = V$, $[D_{2t}u = V]^0_{i,j}$ to obtain a special formula for $u^1_{i,j}$ at the interior mesh points:

$$\begin{aligned} u^1 = &u^0_{i,j} + \Delta t V_{i,j} + \\ &\frac{1}{2}C_x^2(u^0_{i+1,j} - 2u^0_{i,j} + u^0_{i-1,j}) + \frac{1}{2}C_y^2(u^0_{i,j+1} - 2u^0_{i,j} + u^0_{i,j-1}) + \frac{1}{2}\Delta t^2 f^n_{i,j}, \end{aligned} \tag{111}$$

The algorithm is very similar to the one in 1D:

1. Set initial condition $u^0_{i,j} = I(x_i, y_j)$
2. Compute $u^1_{i,j}$ from (111)
3. Set $u^1_{i,j} = 0$ for the boundaries $i = 0, N_x$, $j = 0, N_y$
4. For $n = 1, 2, \ldots, N_t$:
   (a) Find $u^{n+1}_{i,j}$ from (111) for all internal mesh points, $i \in \mathcal{I}^i_x$, $j \in \mathcal{I}^i_y$
   (b) Set $u^{n+1}_{i,j} = 0$ for the boundaries $i = 0, N_x$, $j = 0, N_y$

12.1 Scalar computations

The `solver` function for a 2D case with constant wave velocity and $u = 0$ as boundary condition follows the setup from the similar function for the 1D case in `wave1D_u0.py`, but there are a few necessary extensions. The code is in the program `wave2D_u0.py`.

Domain and mesh. The spatial domain is now $[0, L_x] \times [0, L_y]$, specified by the arguments `Lx` and `Ly`. Similarly, the number of mesh points in the x and y directions, N_x and N_y, become the arguments `Nx` and `Ny`. In multi-dimensional problems it makes less sense to specify a Courant number as the wave velocity is a vector and the mesh spacings may differ in the various spatial directions. We therefore give Δt explicitly. The signature of the `solver` function is then

```
def solver(I, V, f, c, Lx, Ly, Nx, Ny, dt, T,
           user_action=None, version='scalar'):
```

Key parameters used in the calculations are created as

```
x = linspace(0, Lx, Nx+1)                  # mesh points in x dir
y = linspace(0, Ly, Ny+1)                  # mesh points in y dir
dx = x[1] - x[0]
dy = y[1] - y[0]
Nt = int(round(T/float(dt)))
t = linspace(0, N*dt, N+1)                 # mesh points in time
Cx2 = (c*dt/dx)**2;  Cy2 = (c*dt/dy)**2    # help variables
dt2 = dt**2
```

Solution arrays. We store $u^{n+1}_{i,j}$, $u^n_{i,j}$, and $u^{n-1}_{i,j}$ in three two-dimensional arrays,

```
u   = zeros((Nx+1,Ny+1))   # solution array
u_1 = zeros((Nx+1,Ny+1))   # solution at t-dt
u_2 = zeros((Nx+1,Ny+1))   # solution at t-2*dt
```

where $u^{n+1}_{i,j}$ corresponds to `u[i,j]`, $u^n_{i,j}$ to `u_1[i,j]`, and $u^{n-1}_{i,j}$ to `u_2[i,j]`

Index sets. It is also convenient to introduce the index sets (cf. Section 6.4)

```
Ix = range(0, u.shape[0])
Iy = range(0, u.shape[1])
It = range(0, t.shape[0])
```

Computing the solution. Inserting the initial condition `I` in `u_1` and making a callback to the user in terms of the `user_action` function is a straightforward generalization of the 1D code from Section 1.6:

```
for i in Ix:
    for j in Iy:
        u_1[i,j] = I(x[i], y[j])

if user_action is not None:
    user_action(u_1, x, xv, y, yv, t, 0)
```

The `user_action` function has additional arguments compared to the 1D case. The arguments `xv` and `yv` fact will be commented upon in Section 12.2.

The key finite difference formula (103) for updating the solution at a time level is implemented in a separate function as

```
def advance_scalar(u, u_1, u_2, f, x, y, t, n, Cx2, Cy2, dt2,
                   V=None, step1=False):
    Ix = range(0, u.shape[0]);  Iy = range(0, u.shape[1])
    if step1:
        dt = sqrt(dt2)  # save
        Cx2 = 0.5*Cx2;  Cy2 = 0.5*Cy2; dt2 = 0.5*dt2  # redefine
        D1 = 1;  D2 = 0
    else:
        D1 = 2;  D2 = 1
    for i in Ix[1:-1]:
        for j in Iy[1:-1]:
            u_xx = u_1[i-1,j] - 2*u_1[i,j] + u_1[i+1,j]
            u_yy = u_1[i,j-1] - 2*u_1[i,j] + u_1[i,j+1]
            u[i,j] = D1*u_1[i,j] - D2*u_2[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f(x[i], y[j], t[n])
            if step1:
                u[i,j] += dt*V(x[i], y[j])
    # Boundary condition u=0
    j = Iy[0]
    for i in Ix: u[i,j] = 0
    j = Iy[-1]
    for i in Ix: u[i,j] = 0
    i = Ix[0]
    for j in Iy: u[i,j] = 0
    i = Ix[-1]
    for j in Iy: u[i,j] = 0
    return u
```

The `step1` variable has been introduced to allow the formula to be reused for first step $u^1_{i,j}$:

```
u = advance_scalar(u, u_1, u_2, f, x, y, t,
                   n, Cx2, Cy2, dt, V, step1=True)
```

Below, we will make many alternative implementations of the `advance_scalar` function to speed up the code since most of the CPU time in simulations is spent in this function.

Finally, we remark that the `solver` function in the `wave2D_u0.py` code updates arrays for the next time step by switching references as described in Section 4.5. If the solution `u` is return from `solver`, which it is not, it is important to set `u = u_1` after the time loop, otherwise `u` actually contains `u_2`.

12.2 Vectorized computations

The scalar code above turns out to be extremely slow for large 2D meshes, and probably useless in 3D beyond debugging of small test cases. Vectorization is therefore a must for multi-dimensional finite difference computations in Python. For example, with a mesh consisting of 30×30 cells, vectorization brings down the CPU time by a factor of 70 (!).

In the vectorized case we must be able to evaluate user-given functions like $I(x,y)$ and $f(x,y,t)$, provided as Python functions `I(x,y)` and `f(x,y,t)`, for the entire mesh in one array operation. Having the one-dimensional coordinate arrays `x` and `y` is not sufficient: these must be extended to vectorized versions,

```
from numpy import newaxis
xv = x[:,newaxis]
yv = y[newaxis,:]
# or
xv = x.reshape((x.size, 1))
yv = y.reshape((1, y.size))
```

This is a standard required technique when evaluating functions over a 2D mesh, say `sin(xv)*cos(xv)`, which then gives a result with shape `(Nx+1,Ny+1)`.

With the `xv` and `yv` arrays for vectorized computing, setting the initial condition is just a matter of

```
u_1[:,:] = I(xv, yv)
```

One could also have written `u_1 = I(xv, yv)` and let `u_1` point to a new object, but vectorized operations often makes use of direct insertion in the original array through `u_1[:,:]` because sometimes not all of the array is to be filled by such a function evaluation. This is the case with the computational scheme for $u^{n+1}_{i,j}$:

```
def advance_vectorized(u, u_1, u_2, f_a, Cx2, Cy2, dt2,
                       V=None, step1=False):
    if step1:
        dt = sqrt(dt2)  # save
        Cx2 = 0.5*Cx2;  Cy2 = 0.5*Cy2; dt2 = 0.5*dt2  # redefine
        D1 = 1;  D2 = 0
    else:
        D1 = 2;  D2 = 1
    u_xx = u_1[:-2,1:-1] - 2*u_1[1:-1,1:-1] + u_1[2:,1:-1]
    u_yy = u_1[1:-1,:-2] - 2*u_1[1:-1,1:-1] + u_1[1:-1,2:]
    u[1:-1,1:-1] = D1*u_1[1:-1,1:-1] - D2*u_2[1:-1,1:-1] + \
                   Cx2*u_xx + Cy2*u_yy + dt2*f_a[1:-1,1:-1]
    if step1:
        u[1:-1,1:-1] += dt*V[1:-1, 1:-1]
    # Boundary condition u=0
    j = 0
    u[:,j] = 0
    j = u.shape[1]-1
    u[:,j] = 0
    i = 0
    u[i,:] = 0
    i = u.shape[0]-1
    u[i,:] = 0
    return u
```

Array slices in 2D are more complicated to understand than those in 1D, but the logic from 1D applies to each dimension separately. For example, when doing $u^n_{i,j} - u^n_{i-1,j}$ for $i \in \mathcal{I}^+_x$, we just keep `j` constant and make a slice in the first index: `u_1[1:,j] - u_1[:-1,j]`, exactly as in 1D. The `1:` slice specifies all the indices $i = 1, 2, \ldots, N_x$ (up to the last valid index), while `:-1` specifies the

relevant indices for the second term: $0, 1, \ldots, N_x - 1$ (up to, but not including the last index).

In the above code segment, the situation is slightly more complicated, because each displaced slice in one direction is accompanied by a `1:-1` slice in the other direction. The reason is that we only work with the internal points for the index that is kept constant in a difference.

The boundary conditions along the four sides makes use of a slice consisting of all indices along a boundary:

```
u[: ,0] = 0
u[:,Ny] = 0
u[0 ,:] = 0
u[Nx,:] = 0
```

The `f` function is in the above vectorized update of `u` first computed as an array over all mesh points:

```
f_a = f(xv, yv, t[n])
```

We could, alternatively, used the call `f(xv, yv, t[n])[1:-1,1:-1]` in the last term of the update statement, but other implementations in compiled languages benefit from having `f` available in an array rather than calling our Python function `f(x,y,t)` for every point.

Also in the `advance_vectorized` function we have introduced a boolean `step1` to reuse the formula for the first time step in the same way as we did with `advance_scalar`. We refer to the `solver` function in `wave2D_u0.py` for the details on how the overall algorithm is implemented.

The callback function now has the arguments `u, x, xv, y, yv, t, n`. The inclusion of `xv` and `yv` makes it easy to, e.g., compute an exact 2D solution in the callback function and compute errors, through an expression like `u - u_exact(xv, yv, t[n])`.

12.3 Verification

Testing a quadratic solution. The 1D solution from Section 2.4 can be generalized to multi-dimensions and provides a test case where the exact solution also fulfills the discrete equations such that we know (to machine precision) what numbers the solver function should produce. In 2D we use the following generalization of (30):

$$u_{\rm e}(x,y,t) = x(L_x - x)y(L_y - y)(1 + \frac{1}{2}t)\,. \tag{112}$$

This solution fulfills the PDE problem if $I(x,y) = u_{\rm e}(x,y,0)$, $V = \frac{1}{2}u_{\rm e}(x,y,0)$, and $f = 2c^2(1 + \frac{1}{2}t)(y(L_y - y) + x(L_x - x))$. To show that $u_{\rm e}$ also solves the discrete equations, we start with the general results $[D_tD_t1]^n = 0$, $[D_tD_tt]^n = 0$, and $[D_tD_tt^2] = 2$, and use these to compute

$$[D_xD_xu_{\rm e}]^n_{i,j} = [y(L_y - y)(1 + \frac{1}{2}t)D_xD_xx(L_x - x)]^n_{i,j} = y_j(L_y - y_j)(1 + \frac{1}{2}t_n)2\,.$$

A similar calculation must be carried out for the $[D_yD_yu_{\rm e}]^n_{i,j}$ and $[D_tD_tu_{\rm e}]^n_{i,j}$ terms. One must also show that the quadratic solution fits the special formula for $u^1_{i,j}$. The details are left as Exercise 11. The `test_quadratic` function in the `wave2D_u0.py` program implements this verification as a nose test.

13 Migrating loops to Cython

Although vectorization can bring down the CPU time dramatically compared with scalar code, there is still some factor 5-10 to win in these types of applications by implementing the finite difference scheme in compiled code, typically in Fortran, C, or C++. This can quite easily be done by adding a little extra code to our program. Cython is an extension of Python that offers the easiest way to nail our Python loops in the scalar code down to machine code and the efficiency of C.

Cython can be viewed as an extended Python language where variables are declared with types and where functions are marked to be implemented in C. Migrating Python code to Cython is done by copying the desired code segments to functions (or classes) and placing them in one or more separate files with extension `.pyx`.

13.1 Declaring variables and annotating the code

Our starting point is the plain `advance_scalar` function for a scalar implementation of the updating algorithm for new values $u^{n+1}_{i,j}$:

```
def advance_scalar(u, u_1, u_2, f, x, y, t, n, Cx2, Cy2, dt2,
                   V=None, step1=False):
    Ix = range(0, u.shape[0]);  Iy = range(0, u.shape[1])
    if step1:
        dt = sqrt(dt2)  # save
        Cx2 = 0.5*Cx2;  Cy2 = 0.5*Cy2; dt2 = 0.5*dt2  # redefine
        D1 = 1;  D2 = 0
    else:
        D1 = 2;  D2 = 1
    for i in Ix[1:-1]:
        for j in Iy[1:-1]:
            u_xx = u_1[i-1,j] - 2*u_1[i,j] + u_1[i+1,j]
            u_yy = u_1[i,j-1] - 2*u_1[i,j] + u_1[i,j+1]
            u[i,j] = D1*u_1[i,j] - D2*u_2[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f(x[i], y[j], t[n])
            if step1:
                u[i,j] += dt*V(x[i], y[j])
    # Boundary condition u=0
    j = Iy[0]
    for i in Ix: u[i,j] = 0
    j = Iy[-1]
    for i in Ix: u[i,j] = 0
    i = Ix[0]
    for j in Iy: u[i,j] = 0
    i = Ix[-1]
    for j in Iy: u[i,j] = 0
    return u
```

We simply take a copy of this function and put it in a file `wave2D_u0_loop_cy.pyx`. The relevant Cython implementation arises from declaring variables with types and adding some important annotations to speed up array computing in Cython. Let us first list the complete code in the `.pyx` file:

```
import numpy as np
cimport numpy as np
cimport cython
ctypedef np.float64_t DT    # data type

@cython.boundscheck(False)  # turn off array bounds check
@cython.wraparound(False)   # turn off negative indices (u[-1,-1])
cpdef advance(
    np.ndarray[DT, ndim=2, mode='c'] u,
    np.ndarray[DT, ndim=2, mode='c'] u_1,
    np.ndarray[DT, ndim=2, mode='c'] u_2,
    np.ndarray[DT, ndim=2, mode='c'] f,
    double Cx2, double Cy2, double dt2):

    cdef:
        int Ix_start = 0
        int Iy_start = 0
        int Ix_end = u.shape[0]-1
        int Iy_end = u.shape[1]-1
        int i, j
        double u_xx, u_yy

    for i in range(Ix_start+1, Ix_end):
        for j in range(Iy_start+1, Iy_end):
            u_xx = u_1[i-1,j] - 2*u_1[i,j] + u_1[i+1,j]
            u_yy = u_1[i,j-1] - 2*u_1[i,j] + u_1[i,j+1]
            u[i,j] = 2*u_1[i,j] - u_2[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f[i,j]
    # Boundary condition u=0
    j = Iy_start
    for i in range(Ix_start, Ix_end+1): u[i,j] = 0
    j = Iy_end
    for i in range(Ix_start, Ix_end+1): u[i,j] = 0
    i = Ix_start
    for j in range(Iy_start, Iy_end+1): u[i,j] = 0
    i = Ix_end
    for j in range(Iy_start, Iy_end+1): u[i,j] = 0
    return u
```

This example may act as a recipe on how to transform array-intensive code with loops into Cython.

1. Variables are declared with types: for example, `double v` in the argument list instead of just `v`, and `cdef double v` for a variable `v` in the body of the function. A Python `float` object is declared as `double` for translation to C by Cython, while an `int` object is declared by `int`.

2. Arrays need a comprehensive type declaration involving

   - the type `np.ndarray`,
   - the data type of the elements, here 64-bit floats, abbreviated as `DT` through `ctypedef np.float64_t DT` (instead of `DT` we could use the

full name of the data type: `np.float64_t`, which is a Cython-defined type),

- the dimensions of the array, here `ndim=2` and `ndim=1`,
- specification of contiguous memory for the array (`mode='c'`).

3. Functions declared with `cpdef` are translated to C but also accessible from Python.

4. In addition to the standard `numpy` import we also need a special Cython import of `numpy`: `cimport numpy as np`, to appear *after* the standard import.

5. By default, array indices are checked to be within their legal limits. To speed up the code one should turn off this feature for a specific function by placing `@cython.boundscheck(False)` above the function header.

6. Also by default, array indices can be negative (counting from the end), but this feature has a performance penalty and is therefore here turned off by writing `@cython.wraparound(False)` right above the function header.

7. The use of index sets `Ix` and `Iy` in the scalar code cannot be successfully translated to C. One reason is that constructions like `Ix[1:-1]` involve negative indices, and these are now turned off. Another reason is that Cython loops must take the form `for i in xrange` or `for i in range` for being translated into efficient C loops. We have therefore introduced `Ix_start` as `Ix[0]` and `Ix_end` as `Ix[-1]` to hold the start and end of the values of index i. Similar variables are introduced for the j index. A loop `for i in Ix` is with these new variables written as `for i in range(Ix_start, Ix_end+1)`.

Array declaration syntax in Cython.

We have used the syntax `np.ndarray[DT, ndim=2, mode='c']` to declare `numpy` arrays in Cython. There is a simpler, alternative syntax, employing typed memory views, where the declaration looks like `double [:,:]`. However, the full support for this functionality is not yet ready, and in this text we use the full array declaration syntax.

13.2 Visual inspection of the C translation

Cython can visually explain how successfully it can translate a code from Python to C. The command

```
Terminal> cython -a wave2D_u0_loop_cy.pyx
```

produces an HTML file `wave2D_u0_loop_cy.html`, which can be loaded into a web browser to illustrate which lines of the code that have been translated to C. Figure 8 shows the illustrated code. Yellow lines indicate the lines that Cython did not manage to translate to efficient C code and that remain in Python. For the present code we see that Cython is able to translate all the loops with array computing to C, which is our primary goal.

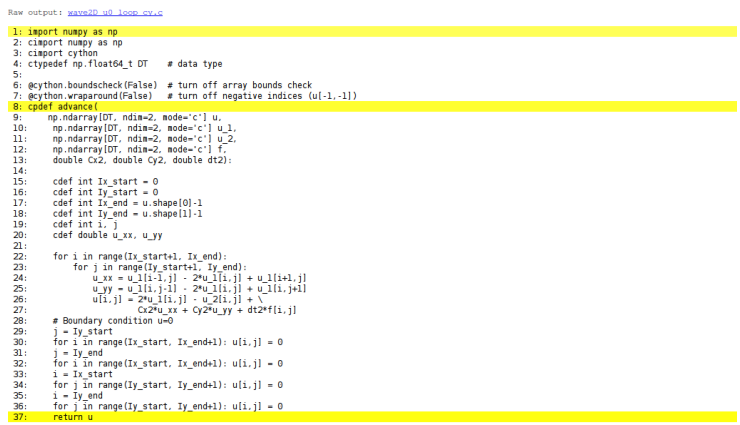

Figure 8: Visual illustration of Cython's ability to translate Python to C.

You can also inspect the generated C code directly, as it appears in the file `wave2D_u0_loop_cy.c`. Nevertheless, understanding this C code requires some familiarity with writing Python extension modules in C by hand. Deep down in the file we can see in detail how the compute-intensive statements are translated some complex C code that is quite different from what we a human would write (at least if a direct correspondence to the mathematics was in mind).

13.3 Building the extension module

Cython code must be translated to C, compiled, and linked to form what is known in the Python world as a *C extension module*. This is usually done by making a `setup.py` script, which is the standard way of building and installing Python software. For an extension module arising from Cython code, the following `setup.py` script is all we need to build and install the module:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

cymodule = 'wave2D_u0_loop_cy'
```

```
setup(
  name=cymodule
  ext_modules=[Extension(cymodule, [cymodule + '.pyx'],)],
  cmdclass={'build_ext': build_ext},
)
```

We run the script by

```
Terminal> python setup.py build_ext --inplace
```

The `-inplace` option makes the extension module available in the current directory as the file `wave2D_u0_loop_cy.so`. This file acts as a normal Python module that can be imported and inspected:

```
>>> import wave2D_u0_loop_cy
>>> dir(wave2D_u0_loop_cy)
['__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '__test__', 'advance', 'np']
```

The important output from the `dir` function is our Cython function `advance` (the module also features the imported `numpy` module under the name `np` as well as many standard Python objects with double underscores in their names).

The `setup.py` file makes use of the `distutils` package in Python and Cython's extension of this package. These tools know how Python was built on the computer and will use compatible compiler(s) and options when building other code in Cython, C, or C++. Quite some experience with building large program systems is needed to do the build process manually, so using a `setup.py` script is strongly recommended.

Simplified build of a Cython module.

When there is no need to link the C code with special libraries, Cython offers a shortcut for generating and importing the extension module:

```
import pyximport; pyximport.install()
```

This makes the `setup.py` script redundant. However, in the `wave2D_u0.py` code we do not use `pyximport` and require an explicit build process of this and many other modules.

13.4 Calling the Cython function from Python

The `wave2D_u0_loop_cy` module contains our `advance` function, which we now may call from the Python program for the wave equation:

```
import wave2D_u0_loop_cy
advance = wave2D_u0_loop_cy.advance
...
for n in It[1:-1:                  # time loop
    f_a[:,:] = f(xv, yv, t[n])     # precompute, size as u
    u = advance(u, u_1, u_2, f_a, x, y, t, Cx2, Cy2, dt2)
```

Efficiency. For a mesh consisting of 120×120 cells, the scalar Python code require 1370 CPU time units, the vectorized version requires 5.5, while the Cython version requires only 1! For a smaller mesh with 60×60 cells Cython is about 1000 times faster than the scalar Python code, and the vectorized version is about 6 times slower than the Cython version.

14 Migrating loops to Fortran

Instead of relying on Cython's (excellent) ability to translate Python to C, we can invoke a compiled language directly and write the loops ourselves. Let us start with Fortran 77, because this is a language with more convenient array handling than C (or plain C++). Or more precisely, we can with ease program with the same multi-dimensional indices in the Fortran code as in the `numpy` arrays in the Python code, while in C these arrays are one-dimensional and requires us to reduce multi-dimensional indices to a single index.

14.1 The Fortran subroutine

We write a Fortran subroutine `advance` in a file `wave2D_u0_loop_f77.f` for implementing the updating formula (111) and setting the solution to zero at the boundaries:

```
      subroutine advance(u, u_1, u_2, f, Cx2, Cy2, dt2, Nx, Ny)
      integer Nx, Ny
      real*8 u(0:Nx,0:Ny), u_1(0:Nx,0:Ny), u_2(0:Nx,0:Ny)
      real*8 f(0:Nx,0:Ny), Cx2, Cy2, dt2
      integer i, j
      real*8 u_xx, u_yy
Cf2py intent(in, out) u

C     Scheme at interior points
      do j = 1, Ny-1
         do i = 1, Nx-1
            u_xx = u_1(i-1,j) - 2*u_1(i,j) + u_1(i+1,j)
            u_yy = u_1(i,j-1) - 2*u_1(i,j) + u_1(i,j+1)
            u(i,j) = 2*u_1(i,j) - u_2(i,j) + Cx2*u_xx + Cy2*u_yy +
     &               dt2*f(i,j)
         end do
      end do

C     Boundary conditions
      j = 0
      do i = 0, Nx
```

```
         u(i,j) = 0
      end do
      j = Ny
      do i = 0, Nx
         u(i,j) = 0
      end do
      i = 0
      do j = 0, Ny
         u(i,j) = 0
      end do
      i = Nx
      do j = 0, Ny
         u(i,j) = 0
      end do
      return
      end
```

This code is plain Fortran 77, except for the special `Cf2py` comment line, which here specifies that `u` is both an input argument *and* an object to be returned from the `advance` routine. Or more precisely, Fortran is not able return an array from a function, but we need a *wrapper code* in C for the Fortran subroutine to enable calling it from Python, and in this wrapper code one can return `u` to the calling Python code.

Remark.

It is not strictly necessary to return `u` to the calling Python code since the `advance` function will modify the elements of `u`, but the convention in Python is to get all output from a function as returned values. That is, the right way of calling the above Fortran subroutine from Python is

```
u = advance(u, u_1, u_2, f, Cx2, Cy2, dt2)
```

The less encouraged style, which works and resembles the way the Fortran subroutine is called from Fortran, reads

```
advance(u, u_1, u_2, f, Cx2, Cy2, dt2)
```

14.2 Building the Fortran module with f2py

The nice feature of writing loops in Fortran is that the tool `f2py` can with very little work produce a C extension module such that we can call the Fortran version of `advance` from Python. The necessary commands to run are

```
Terminal> f2py -m wave2D_u0_loop_f77 -h wave2D_u0_loop_f77.pyf \
          --overwrite-signature wave2D_u0_loop_f77.f
```

```
Terminal> f2py -c wave2D_u0_loop_f77.pyf --build-dir build_f77 \
          -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_f77.f
```

The first command asks `f2py` to interpret the Fortran code and make a Fortran 90 specification of the extension module in the file `wave2D_u0_loop_f77.pyf`. The second command makes `f2py` generate all necessary wrapper code, compile our Fortran file and the wrapper code, and finally build the module. The build process takes place in the specified subdirectory `build_f77` so that files can be inspected if something goes wrong. The option `-DF2PY_REPORT_ON_ARRAY_COPY=1` makes `f2py` write a message for every array that is copied in the communication between Fortran and Python, which is very useful for avoiding unnecessary array copying (see below). The name of the module file is `wave2D_u0_loop_f77.so`, and this file can be imported and inspected as any other Python module:

```
>>> import wave2D_u0_loop_f77
>>> dir(wave2D_u0_loop_f77)
['__doc__', '__file__', '__name__', '__package__',
 '__version__', 'advance']
>>> print wave2D_u0_loop_f77.__doc__
This module 'wave2D_u0_loop_f77' is auto-generated with f2py....
Functions:
  u = advance(u,u_1,u_2,f,cx2,cy2,dt2,
      nx=(shape(u,0)-1),ny=(shape(u,1)-1))
```

Examine the doc strings!

Printing the doc strings of the module and its functions is extremely important after having created a module with `f2py`, because `f2py` makes Python interfaces to the Fortran functions that are different from how the functions are declared in the Fortran code (!). The rationale for this behavior is that `f2py` creates *Pythonic* interfaces such that Fortran routines can be called in the same way as one calls Python functions. Output data from Python functions is always returned to the calling code, but this is technically impossible in Fortran. Also, arrays in Python are passed to Python functions without their dimensions because that information is packed with the array data in the array objects, but this is not possible in Fortran. Therefore, `f2py` removes array dimensions from the argument list, and `f2py` makes it possible to return objects back to Python.

Let us follow the advice of examining the doc strings and take a close look at the documentation `f2py` has generated for our Fortran `advance` subroutine:

```
>>> print wave2D_u0_loop_f77.advance.__doc__
This module 'wave2D_u0_loop_f77' is auto-generated with f2py
Functions:
  u = advance(u,u_1,u_2,f,cx2,cy2,dt2,
              nx=(shape(u,0)-1),ny=(shape(u,1)-1))
```

```
.
advance - Function signature:
  u = advance(u,u_1,u_2,f,cx2,cy2,dt2,[nx,ny])
Required arguments:
  u : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  u_1 : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  u_2 : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  f : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  cx2 : input float
  cy2 : input float
  dt2 : input float
Optional arguments:
  nx := (shape(u,0)-1) input int
  ny := (shape(u,1)-1) input int
Return objects:
  u : rank-2 array('d') with bounds (nx + 1,ny + 1)
```

Here we see that the `nx` and `ny` parameters declared in Fortran are optional arguments that can be omitted when calling `advance` from Python.

We strongly recommend to print out the documentation of *every* Fortran function to be called from Python and make sure the call syntax is exactly as listed in the documentation.

14.3 How to avoid array copying

Multi-dimensional arrays are stored as a stream of numbers in memory. For a two-dimensional array consisting of rows and columns there are two ways of creating such a stream: *row-major ordering*, which means that rows are stored consecutively in memory, or *column-major ordering*, which means that the columns are stored one after each other. All programming languages inherited from C, including Python, apply the row-major ordering, but Fortran uses column-major storage. Thinking of a two-dimensional array in Python or C as a matrix, it means that Fortran works with the transposed matrix.

Fortunately, `f2py` creates extra code so that accessing `u(i,j)` in the Fortran subroutine corresponds to the element `u[i,j]` in the underlying `numpy` array (without the extra code, `u(i,j)` in Fortran would access `u[j,i]` in the `numpy` array). Technically, `f2py` takes a copy of our `numpy` array and reorders the data before sending the array to Fortran. Such copying can be costly. For 2D wave simulations on a 60×60 grid the overhead of copying is a factor of 5, which means that almost the whole performance gain of Fortran over vectorized `numpy` code is lost!

To avoid having `f2py` to copy arrays with C storage to the corresponding Fortran storage, we declare the arrays with Fortran storage:

```
order = 'Fortran' if version == 'f77' else 'C'
u   = zeros((Nx+1,Ny+1), order=order)   # solution array
u_1 = zeros((Nx+1,Ny+1), order=order)   # solution at t-dt
u_2 = zeros((Nx+1,Ny+1), order=order)   # solution at t-2*dt
```

In the compile and build step of using `f2py`, it is recommended to add an extra option for making `f2py` report on array copying:

```
Terminal> f2py -c wave2D_u0_loop_f77.pyf --build-dir build_f77 \
          -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_f77.f
```

It can sometimes be a challenge to track down which array that causes a copying. There are two principal reasons for copying array data: either the array does not have Fortran storage or the element types do not match those declared in the Fortran code. The latter cause is usually effectively eliminated by using `real*8` data in the Fortran code and `float64` (the default `float` type in `numpy`) in the arrays on the Python side. The former reason is more common, and to check whether an array before a Fortran call has the right storage one can print the result of `isfortran(a)`, which is `True` if the array `a` has Fortran storage.

Let us look at an example where we face problems with array storage. A typical problem in the `wave2D_u0.py` code is to set

```
f_a = f(xv, yv, t[n])
```

before the call to the Fortran `advance` routine. This computation creates a new array with C storage. An undesired copy of `f_a` will be produced when sending `f_a` to a Fortran routine. There are two remedies, either direct insertion of data in an array with Fortran storage,

```
f_a = zeros((Nx+1, Ny+1), order='Fortran')
...
f_a[:,:] = f(xv, yv, t[n])
```

or remaking the `f(xv, yv, t[n])` array,

```
f_a = asarray(f(xv, yv, t[n]), order='Fortran')
```

The former remedy is most efficient if the `asarray` operation is to be performed a large number of times.

Efficiency. The efficiency of this Fortran code is very similar to the Cython code. There is usually nothing more to gain, from a computational efficiency point of view, by implementing the *complete* Python program in Fortran or C. That will just be a lot more code for all administering work that is needed in scientific software, especially if we extend our sample program `wave2D_u0.py` to handle a real scientific problem. Then only a small portion will consist of loops with intensive array calculations. These can be migrated to Cython or Fortran as explained, while the rest of the programming can be more conveniently done in Python.

15 Migrating loops to C via Cython

The computationally intensive loops can alternatively be implemented in C code. Just as Fortran calls for care regarding the storage of two-dimensional

arrays, working with two-dimensional arrays in C is a bit tricky. The reason is that `numpy` arrays are viewed as one-dimensional arrays when transferred to C, while C programmers will think of `u`, `u_1`, and `u_2` as two dimensional arrays and index them like `u[i][j]`. The C code must declare `u` as `double* u` and translate an index pair `[i][j]` to a corresponding single index when `u` is viewed as one-dimensional. This translation requires knowledge of how the numbers in `u` are stored in memory.

15.1 Translating index pairs to single indices

Two-dimensional `numpy` arrays with the default C storage are stored row by row. In general, multi-dimensional arrays with C storage are stored such that the last index has the fastest variation, then the next last index, and so on, ending up with the slowest variation in the first index. For a two-dimensional `u` declared as `zeros((Nx+1,Ny+1))` in Python, the individual elements are stored in the following order:

```
u[0,0], u[0,1], u[0,2], ..., u[0,Ny], u[1,0], u[1,1], ...,
u[1,Ny], u[2,0], ..., u[Nx,0], u[Nx,1], ..., u[Nx, Ny]
```

Viewing `u` as one-dimensional, the index pair (i, j) translates to $i(N_y + 1) + j$. So, where a C programmer would naturally write an index `u[i][j]`, the indexing must read `u[i*(Ny+1) + j]`. This is tedious to write, so it can be handy to define a C macro,

```
#define idx(i,j) (i)*(Ny+1) + j
```

so that we can write `u[idx(i,j)]`, which reads much better and is easier to debug.

Be careful with macro definitions.

Macros just perform simple text substitutions: `idx(hello,world)` is expanded to `(hello)*(Ny+1) + world`. The parenthesis in `(i)` are essential - using the natural mathematical formula `i*(Ny+1) + j` in the macro definition, `idx(i-1,j)` would expand to `i-1*(Ny+1) + j`, which is the wrong formula. Macros are handy, but requires careful use. In C++, inline functions are safer and replace the need for macros.

15.2 The complete C code

The C version of our function `advance` can be coded as follows.

```
#define idx(i,j) (i)*(Ny+1) + j

void advance(double* u, double* u_1, double* u_2, double* f,
```

```
              double Cx2, double Cy2, double dt2, int Nx, int Ny)
{
  int i, j;
  double u_xx, u_yy;
  /* Scheme at interior points */
  for (i=1; i<=Nx-1; i++) {
    for (j=1; j<=Ny-1; j++) {
      u_xx = u_1[idx(i-1,j)] - 2*u_1[idx(i,j)] + u_1[idx(i+1,j)];
      u_yy = u_1[idx(i,j-1)] - 2*u_1[idx(i,j)] + u_1[idx(i,j+1)];
      u[idx(i,j)] = 2*u_1[idx(i,j)] - u_2[idx(i,j)] +
        Cx2*u_xx + Cy2*u_yy + dt2*f[idx(i,j)];
    }
  }
  /* Boundary conditions */
  j = 0;  for (i=0; i<=Nx; i++) u[idx(i,j)] = 0;
  j = Ny; for (i=0; i<=Nx; i++) u[idx(i,j)] = 0;
  i = 0;  for (j=0; j<=Ny; j++) u[idx(i,j)] = 0;
  i = Nx; for (j=0; j<=Ny; j++) u[idx(i,j)] = 0;
}
```

15.3 The Cython interface file

All the code above appears in a file `wave2D_u0_loop_c.c`. We need to compile this file together with C wrapper code such that `advance` can be called from Python. Cython can be used to generate appropriate wrapper code. The relevant Cython code for interfacing C is placed in a file with extension `.pyx`. Here this file, called `wave2D_u0_loop_c_cy.pyx`, looks like

```
import numpy as np
cimport numpy as np
cimport cython

cdef extern from "wave2D_u0_loop_c.h":
    void advance(double* u, double* u_1, double* u_2, double* f,
                 double Cx2, double Cy2, double dt2,
                 int Nx, int Ny)

@cython.boundscheck(False)
@cython.wraparound(False)
def advance_cwrap(
    np.ndarray[double, ndim=2, mode='c'] u,
    np.ndarray[double, ndim=2, mode='c'] u_1,
    np.ndarray[double, ndim=2, mode='c'] u_2,
    np.ndarray[double, ndim=2, mode='c'] f,
    double Cx2, double Cy2, double dt2):
    advance(&u[0,0], &u_1[0,0], &u_2[0,0], &f[0,0],
            Cx2, Cy2, dt2,
            u.shape[0]-1, u.shape[1]-1)
    return u
```

We first declare the C functions to be interfaced. These must also appear in a C header file, `wave2D_u0_loop_c.h`,

```
extern void advance(double* u, double* u_1, double* u_2, double* f,
                    double Cx2, double Cy2, double dt2,
                    int Nx, int Ny);
```

The next step is to write a Cython function with Python objects as arguments. The name `advance` is already used for the C function so the function to be called from Python is named `advance_cwrap`. The contents of this function is simply a call to the `advance` version in C. To this end, the right information from the Python objects must be passed on as arguments to `advance`. Arrays are sent with their C pointers to the first element, obtained in Cython as `&u[0,0]` (the `&` takes the address of a C variable). The `Nx` and `Ny` arguments in `advance` are easily obtained from the shape of the `numpy` array `u`. Finally, `u` must be returned such that we can set `u = advance(...)` in Python.

15.4 Building the extension module

It remains to build the extension module. An appropriate `setup.py` file is

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

sources = ['wave2D_u0_loop_c.c', 'wave2D_u0_loop_c_cy.pyx']
module = 'wave2D_u0_loop_c_cy'
setup(
  name=module,
  ext_modules=[Extension(module, sources,
                         libraries=[], # C libs to link with
                         )],
  cmdclass={'build_ext': build_ext},
)
```

All we need to specify is the `.c` file(s) and the `.pyx` interface file. Cython is automatically run to generate the necessary wrapper code. Files are then compiled and linked to an extension module residing in the file `wave2D_u0_loop_c_cy.so`. Here is a session with running `setup.py` and examining the resulting module in Python

```
Terminal> python setup.py build_ext --inplace
Terminal> python
>>> import wave2D_u0_loop_c_cy as m
>>> dir(m)
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
 '__test__', 'advance_cwrap', 'np']
```

The call to the C version of `advance` can go like this in Python:

```
import wave2D_u0_loop_c_cy
advance = wave2D_u0_loop_c_cy.advance_cwrap
...
```

```
f_a[:,:] = f(xv, yv, t[n])
u = advance(u, u_1, u_2, f_a, Cx2, Cy2, dt2)
```

Efficiency. In this example, the C and Fortran code runs at the same speed, and there are no significant differences in the efficiency of the wrapper code. The overhead implied by the wrapper code is negligible as long as we do not work with very small meshes and consequently little numerical work in the `advance` function.

16 Migrating loops to C via f2py

An alternative to using Cython for interfacing C code is to apply `f2py`. The C code is the same, just the details of specifying how it is to be called from Python differ. The `f2py` tool requires the call specification to be a Fortran 90 module defined in a `.pyf` file. This file was automatically generated when we interfaced a Fortran subroutine. With a C function we need to write this module ourselves, or we can use a trick and let `f2py` generate it for us. The trick consists in writing the signature of the C function with Fortran syntax and place it in a Fortran file, here `wave2D_u0_loop_c_f2py_signature.f`:

```
      subroutine advance(u, u_1, u_2, f, Cx2, Cy2, dt2, Nx, Ny)
Cf2py intent(c) advance
      integer Nx, Ny, N
      real*8 u(0:Nx,0:Ny), u_1(0:Nx,0:Ny), u_2(0:Nx,0:Ny)
      real*8 f(0:Nx, 0:Ny), Cx2, Cy2, dt2
Cf2py intent(in, out) u
Cf2py intent(c) u, u_1, u_2, f, Cx2, Cy2, dt2, Nx, Ny
      return
      end
```

Note that we need a special `f2py` instruction, through a `Cf2py` comment line, for telling that all the function arguments are C variables. We also need to specify that the function is actually in C: `intent(c) advance`.

Since `f2py` is just concerned with the function signature and not the complete contents of the function body, it can easily generate the Fortran 90 module specification based solely on the signature above:

```
Terminal> f2py -m wave2D_u0_loop_c_f2py \
          -h wave2D_u0_loop_c_f2py.pyf --overwrite-signature \
          wave2D_u0_loop_c_f2py_signature.f
```

The compile and build step is as for the Fortran code, except that we list C files instead of Fortran files:

```
Terminal> f2py -c wave2D_u0_loop_c_f2py.pyf \
          --build-dir tmp_build_c \
          -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_c.c
```

As when interfacing Fortran code with `f2py`, we need to print out the doc string to see the exact call syntax from the Python side. This doc string is identical for the C and Fortran versions of `advance`.

16.1 Migrating loops to C++ via f2py

C++ is a much more versatile language than C or Fortran and has over the last two decades become very popular for numerical computing. Many will therefore prefer to migrate compute-intensive Python code to C++. This is, in principle, easy: just write the desired C++ code and use some tool for interfacing it from Python. A tool like SWIG can interpret the C++ code and generate interfaces for a wide range of languages, including Python, Perl, Ruby, and Java. However, SWIG is a comprehensive tool with a correspondingly steep learning curve. Alternative tools, such as Boost Python, SIP, and Shiboken are similarly comprehensive. Simpler tools include PyBindGen,

A technically much easier way of interfacing C++ code is to drop the possibility to use C++ classes directly from Python, but instead make a C interface to the C++ code. The C interface can be handled by `f2py` as shown in the example with pure C code. Such a solution means that classes in Python and C++ cannot be mixed and that only primitive data types like numbers, strings, and arrays can be transferred between Python and C++. Actually, this is often a very good solution because it forces the C++ code to work on array data, which usually gives faster code than if fancy data structures with classes are used. The arrays coming from Python, and looking like plain C/C++ arrays, can be efficiently wrapped in more user-friendly C++ array classes in the C++ code, if desired.

17 Using classes to implement a simulator

- Introduce classes `Mesh`, `Function`, `Problem`, `Solver`, `Visualizer`, `File`

18 Exercises

Exercise 11: Check that a solution fulfills the discrete model

Carry out all mathematical details to show that (112) is indeed a solution of the discrete model for a 2D wave equation with $u = 0$ on the boundary. One must check the boundary conditions, the initial conditions, the general discrete equation at a time level and the special version of this equation for the first time level. Filename: `check_quadratic_solution.pdf`.

Project 12: Calculus with 2D/3D mesh functions

The goal of this project is to redo Project 5 with 2D and 3D mesh functions ($f_{i,j}$ and $_{fi,j,k}$).

Differentiation. The differentiation results in a discrete gradient function, which in the 2D case can be represented by a three-dimensional array `df[d,i,j]` where `d` represents the direction of the derivative and `i` and `j` are mesh point counters in 2D (the 3D counterpart is `df[d,i,j,k]`).

Integration. The integral of a 2D mesh function $f_{i,j}$ is defined as

$$F_{i,j} = \int_{y_0}^{y_j} \int_{x_0}^{x_i} f(x,y)dxdy,$$

where $f(x,y)$ is a function that takes on the values of the discrete mesh function $f_{i,j}$ at the mesh points, but can also be evaluated in between the mesh points. The particular variation between mesh points can be taken as bilinear, but this is not important as we will use a product Trapezoidal rule to approximate the integral over a cell in the mesh and then we only need to evaluate $f(x,y)$ at the mesh points.

Suppose $F_{i,j}$ is computed. The calculation of $F_{i+1,j}$ is then

$$\begin{aligned}
F_{i+1,j} &= F_{i,j} + \int_{x_i}^{x_{i+1}} \int_{y_0}^{y_j} f(x,y)dydx \\
&\approx \Delta x \int_{y_0}^{y_j} f(x_{i+\frac{1}{2}}, y)dy \\
&\approx \Delta x \frac{1}{2} \left(\int_{y_0}^{y_j} f(x_i, y)dy + \int_{y_0}^{y_j} f(x_{i+1}, y)dy \right)
\end{aligned}$$

The integrals in the y direction can be approximated by a Trapezoidal rule. A similar idea can be used to compute $F_{i,j+1}$. Thereafter, $F_{i+1,j+1}$ can be computed by adding the integral over the final corner cell to $F_{i+1,j} + F_{i,j+1} - F_{i,j}$. Carry out the details of these computations and extend the ideas to 3D. Filename: `mesh_calculus_3D.py`.

Exercise 13: Implement Neumann conditions in 2D

Modify the `wave2D_u0.py` program, which solves the 2D wave equation $u_{tt} = c^2(u_{xx} + u_{yy})$ with constant wave velocity c and $u = 0$ on the boundary, to have Neumann boundary conditions: $\partial u/\partial n = 0$. Include both scalar code (for debugging and reference) and vectorized code (for speed).

To test the code, use $u = 1.2$ as solution ($I(x,y) = 1.2$, $V = f = 0$, and c arbitrary), which should be exactly reproduced with any mesh as long as the stability criterion is satisfied. Another test is to use the plug-shaped pulse in the

`pulse` function from Section 8 and the `wave1D_dn_vc.py` program. This pulse is exactly propagated in 1D if $c\Delta t/\Delta x = 1$. Check that also the 2D program can propagate this pulse exactly in x direction ($c\Delta t/\Delta x = 1$, Δy arbitrary) and y direction ($c\Delta t/\Delta y = 1$, Δx arbitrary). Filename: `wave2D_dn.py`.

Exercise 14: Test the efficiency of compiled loops in 3D

Extend the `wave2D_u0.py` code and the Cython, Fortran, and C versions to 3D. Set up an efficiency experiment to determine the relative efficiency of pure scalar Python code, vectorized code, Cython-compiled loops, Fortran-compiled loops, and C-compiled loops. Normalize the CPU time for each mesh by the fastest version. Filename: `wave3D_u0.py`.

19 Applications of wave equations

This section presents a range of wave equation models for different physical phenomena. Although many wave motion problems in physics can be modeled by the standard linear wave equation, or a similar formulation with a system of first-order equations, there are some exceptions. Perhaps the most important is water waves: these are modeled by the Laplace equation with time-dependent boundary conditions at the water surface (long water waves, however, can be approximated by a standard wave equation, see Section 19.7). Quantum mechanical waves constitute another example where the waves are governed by the Schrödinger equation and not a standard wave equation. Many wave phenomena also need to take nonlinear effects into account when the wave amplitude is significant. Shock waves in the air is a primary example.

The derivations in the following are very brief. Those with a firm background in continuum mechanics will probably have enough information to fill in the details, while other readers will hopefully get some impression of the physics and approximations involved when establishing wave equation models.

19.1 Waves on a string

Figure 9 shows a model we may use to derive the equation for waves on a string. The string is modeled as a set of discrete point masses (at mesh points) with elastic strings in between. The strings are at a high constant tension T. We let the mass at mesh point x_i be m_i. The displacement of this mass point in y direction is denoted by $u_i(t)$.

The motion of mass m_i is governed by Newton's second law of motion. The position of the mass at time t is $x_i\boldsymbol{i} + u_i(t)\boldsymbol{j}$, where $\boldsymbol{i}$ and $\boldsymbol{j}$ are unit vectors in the x and y direction, respectively. The acceleration is then $u_i''(t)\boldsymbol{j}$. Two forces are acting on the mass as indicated in Figure 9. The force $\boldsymbol{T}^-$ acting toward the point x_{i-1} can be decomposed as

$$\boldsymbol{T}^- = -T\sin\phi\boldsymbol{i} - T\cos\phi\boldsymbol{j},$$

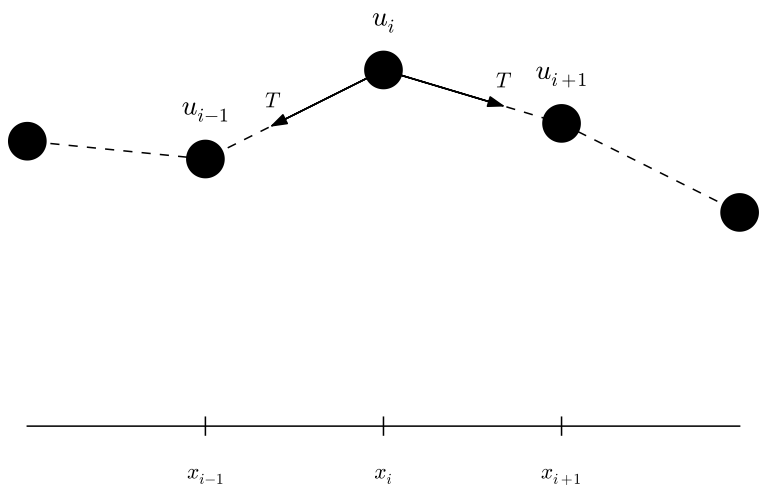

Figure 9: Discrete string model with point masses connected by elastic strings.

where ϕ is the angle between the force and the line $x = x_i$. Let $\Delta u_i = u_i - u_{i-1}$ and let $\Delta s_i = \sqrt{\Delta u_i^2 + (x_i - x_{i-1})^2}$ be the distance from mass m_{i-1} to mass m_i. It is seen that $\cos\phi = \Delta u_i/\Delta s_i$ and $\sin\phi = (x_i - x_{i-1})/\Delta s$ or $\Delta x/\Delta s_i$ if we introduce a constant mesh spacing $\Delta x = x_i - x_{i-1}$. The force can then be written

$$\boldsymbol{T}^- = -T\frac{\Delta x}{\Delta s_i}\boldsymbol{i} - T\frac{\Delta u_i}{\Delta s_i}\boldsymbol{j}\,.$$

The force $\boldsymbol{T}^+$ acting toward x_{i+1} can be calculated in a similar way:

$$\boldsymbol{T}^+ = T\frac{\Delta x}{\Delta s_{i+1}}\boldsymbol{i} + T\frac{\Delta u_{i+1}}{\Delta s_{i+1}}\boldsymbol{j}\,.$$

Newton's second law becomes

$$m_i u_i''(t)\boldsymbol{j} = \boldsymbol{T}^+ + \boldsymbol{T}^-,$$

which gives the component equations

$$T\frac{\Delta x}{\Delta s_i} = T\frac{\Delta x}{\Delta s_{i+1}}, \tag{113}$$

$$m_i u_i''(t) = T\frac{\Delta u_{i+1}}{\Delta s_{i+1}} - T\frac{\Delta u_i}{\Delta s_i}\,. \tag{114}$$

A basic reasonable assumption for a string is small displacements u_i and small displacement gradients $\Delta u_i/\Delta x$. For small $g = \Delta u_i/\Delta x$ we have that

$$\Delta s_i = \sqrt{\Delta u_i^2 + \Delta x^2} = \Delta x\sqrt{1+g^2} + \Delta x(1 + \frac{1}{2}g^2 + \mathcal{O}(g^4) \approx \Delta x\,.$$

Equation (113) is then simply the identity $T = T$, while (114) can be written as

$$m_i u_i''(t) = T\frac{\Delta u_{i+1}}{\Delta x} - T\frac{\Delta u_i}{\Delta x},$$

which upon division by Δx and introducing the density $\varrho_i = m_i/\Delta x$ becomes

$$\varrho_i u_i''(t) = T\frac{1}{\Delta x^2}\left(u_{i+1} - 2u_i + u_{i-1}\right)\,. \tag{115}$$

We can now choose to approximate u_i'' by a finite difference in time and get the discretized wave equation,

$$\varrho_i\frac{1}{\Delta t^2}\left(u_i^{n+1} - 2u_i^n - u_i^{n-1}\right) = T\frac{1}{\Delta x^2}\left(u_{i+1} - 2u_i + u_{i-1}\right)\,. \tag{116}$$

On the other hand, we may go to the continuum limit $\Delta x \to 0$ and replace $u_i(t)$ by $u(x,t)$, ϱ_i by $\varrho(x)$, and recognize that the right-hand side of (115) approaches $\partial^2 u/\partial x^2$ as $\Delta x \to 0$. We end up with the continuous model for waves on a string:

$$\varrho\frac{\partial^2 u}{\partial t^2} = T\frac{\partial^2 u}{\partial x^2}\,. \tag{117}$$

Note that the density ϱ may change along the string, while the tension T is a constant. With variable wave velocity $c(x) = \sqrt{T/\varrho(x)}$ we can write the wave equation in the more standard form

$$\frac{\partial^2 u}{\partial t^2} = c^2(x)\frac{\partial^2 u}{\partial x^2}\,. \tag{118}$$

Because of the way ϱ enters the equations, the variable wave velocity does *not* appear inside the derivatives as in many other versions of the wave equation. However, most strings of interest have constant ϱ.

The end point of a string are fixed so that the displacement u is zero. The boundary conditions are therefore $u = 0$.

Damping. Air resistance and non-elastic effects in the string will contribute to reduce the amplitudes of the waves so that the motion dies out after some time. This damping effect can be modeled by a term bu_t on the left-hand side of the equation

$$\varrho\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = T\frac{\partial^2 u}{\partial x^2}\,. \tag{119}$$

The parameter b must normally be determined from physical experiments.

External forcing. It is easy to include an external force acting on the string. Say we have a vertical force $\tilde{f}_i\boldsymbol{j}$ acting on mass m_i. This force affects the vertical component of Newton's law and gives rise to an extra term $\tilde{f}(x,t)$ on the right-hand side of (117). In the model (118) we would add a term $f(x,t) = \tilde{f}(x,y)/\varrho(x)$.

Modeling the tension via springs. We assumed, in the derivation above, that the tension in the string, T, was constant. It is easy to check this assumption by modeling the string segments between the masses as standard springs, where the force (tension T) is proportional to the elongation of the spring segment. Let k be the spring constant, and set $T_i = k\Delta\ell$ for the tension in the spring segment between x_{i-1} and x_i, where $\Delta\ell$ is the elongation of this segment from the tension-free state. A basic feature of a string is that it has high tension in the equilibrium position $u = 0$. Let the string segment have an elongation $\Delta\ell_0$ in the equilibrium position. After deformation of the string, the elongation is $\Delta\ell = \Delta\ell_0 + \Delta s_i$: $T_i = k(\Delta\ell_0 + \Delta s_i) \approx k(\Delta\ell_0 + \Delta x)$. This shows that T_i is independent of i. Moreover, the extra approximate elongation Δx is very small compared to $\Delta\ell_0$, so we may well set $T_i = T = k\Delta\ell_0$. This means that the tension is completely dominated by the initial tension determined by the tuning of the string. The additional deformations of the spring during the vibrations do not introduce significant changes in the tension.

19.2 Waves on a membrane

19.3 Elastic waves in a rod

Consider an elastic rod subject to a hammer impact at the end. This experiment will give rise to an elastic deformation pulse that travels through the rod. A mathematical model for longitudinal waves along an elastic rod starts with the general equation for deformations and stresses in an elastic medium,

$$\varrho\boldsymbol{u}_{tt} = \nabla\cdot\boldsymbol{\sigma} + \varrho\boldsymbol{f}, \tag{120}$$

where ϱ is the density, $\boldsymbol{u}$ the displacement field, $\boldsymbol{\sigma}$ the stress tensor, and $\boldsymbol{f}$ body forces. The latter has normally no impact on elastic waves.

For stationary deformation of an elastic rod, one has that $\sigma_{xx} = Eu_x$, with all other stress components being zero. Moreover, $\boldsymbol{u} = u(x)\boldsymbol{i}$. The parameter E

is known as Young's modulus. Assuming that this simple stress and deformation field, which is exact in the stationary case, is a good approximation in the transient case with wave motion, (120) simplifies to

$$\varrho \frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(E\frac{\partial u}{\partial x}\right). \tag{121}$$

The associated boundary conditions are u or $\sigma_{xx} = Eu_x$ known, typically $u = 0$ for a clamped end and $\sigma_{xx} = 0$ for a free end.

19.4 The acoustic model for seismic waves

Seismic waves are used to infer properties of subsurface geological structures. The physical model is a heterogeneous elastic medium where sound is propagated by small elastic vibrations. The general mathematical model for deformations in an elastic medium is based on Newton's second law,

$$\varrho \boldsymbol{u}_{tt} = \nabla \cdot \boldsymbol{\sigma} + \varrho \boldsymbol{f}, \tag{122}$$

and a constitutive law relating $\boldsymbol{\sigma}$ to $\boldsymbol{u}$, often Hooke's generalized law,

$$\boldsymbol{\sigma} = K\nabla \cdot \boldsymbol{u}\,\boldsymbol{I} + G(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T - \frac{2}{3}\nabla \cdot \boldsymbol{u}\,\boldsymbol{I})\,. \tag{123}$$

Here, $\boldsymbol{u}$ is the displacement field, $\boldsymbol{\sigma}$ is the stress tensor, $\boldsymbol{I}$ is the identity tensor, ϱ is the medium's density, $\boldsymbol{f}$ are body forces (such as gravity), K is the medium's bulk modulus and G is the shear modulus. All these quantities may vary in space, while $\boldsymbol{u}$ and $\boldsymbol{\sigma}$ will also show significant variation in time during wave motion.

The acoustic approximation to elastic waves arises from a basic assumption that the second term in Hooke's law, representing the deformations that give rise to shear stresses, can be neglected. This assumption can be interpreted as approximating the geological medium by a fluid. Neglecting also the body forces $\boldsymbol{f}$, (122) becomes

$$\varrho \boldsymbol{u}_{tt} = \nabla(K\nabla \cdot \boldsymbol{u}) \tag{124}$$

Introducing p as a pressure via

$$p = -K\nabla \cdot \boldsymbol{u}, \tag{125}$$

and dividing (124) by ϱ, we get

$$\boldsymbol{u}_{tt} = -\frac{1}{\varrho}\nabla p\,. \tag{126}$$

Taking the divergence of this equation, using $\nabla \cdot \boldsymbol{u} = -p/K$ from (125), gives the *acoustic approximation to elastic waves*:

$$p_{tt} = K\nabla \cdot \left(\frac{1}{\varrho}\nabla p\right). \tag{127}$$

This is a standard, linear wave equation with variable coefficients. It is common to add a source term $s(x, y, z, t)$ to model the generation of sound waves:

$$p_{tt} = K\nabla \cdot \left(\frac{1}{\varrho}\nabla p\right) + s\,. \tag{128}$$

A common additional approximation of (128) is based on using the chain rule on the right-hand side,

$$K\nabla \cdot \left(\frac{1}{\varrho}\nabla p\right) = \frac{K}{\varrho}\nabla^2 p + K\nabla\left(\frac{1}{\varrho}\right)\cdot\nabla p \approx \frac{K}{\varrho}\nabla^2 p,$$

under the assumption that the relative spatial gradient $\nabla\varrho^{-1} = -\varrho^{-2}\nabla\varrho$ is small. This approximation results in the simplified equation

$$p_{tt} = \frac{K}{\varrho}\nabla^2 p + s\,. \tag{129}$$

The acoustic approximations to seismic waves are used for sound waves in the ground, and the Earth's surface is then a boundary where p equals the atmospheric pressure p_0 such that the boundary condition becomes $p = p_0$.

Anisotropy. Quite often in geological materials, the effective wave velocity $c = \sqrt{K/\varrho}$ is different in different spatial directions because geological layers are compacted such that the properties in the horizontal and vertical direction differ. With z as the vertical coordinate, we can introduce a vertical wave velocity c_z and a horizontal wave velocity c_h, and generalize (129) to

$$p_{tt} = c_z^2 p_{zz} + c_h^2(p_{xx} + p_{yy}) + s\,. \tag{130}$$

19.5 Sound waves in liquids and gases

Sound waves arise from pressure and density variations in fluids. The starting point of modeling sound waves is the basic equations for a compressible fluid where we omit viscous (frictional) forces, body forces (gravity, for instance), and temperature effects:

$$\varrho_t + \nabla\cdot(\varrho\boldsymbol{u}) = 0, \tag{131}$$

$$\varrho\boldsymbol{u}_t + \varrho\boldsymbol{u}\cdot\nabla\boldsymbol{u} = -\nabla p, \tag{132}$$

$$\varrho = \varrho(p)\,. \tag{133}$$

These equations are often referred to as the Euler equations for the motion of a fluid. The parameters involved are the density ϱ, the velocity $\boldsymbol{u}$, and the pressure p. Equation (132) reflects mass balance, (131) is Newton's second law for a fluid, with frictional and body forces omitted, and (133) is a constitutive law relating density to pressure by thermodynamics considerations. A typical

model for (133) is the so-called isentropic relation, valid for adiabatic processes where there is no heat transfer:

$$\varrho = \varrho_0 \left(\frac{p}{p_0}\right)^{1/\gamma} . \tag{134}$$

Here, p_0 and ϱ_0 are references values for p and ϱ when the fluid is at rest, and γ is the ratio of specific heat at constant pressure and constant volume ($\gamma = 5/3$ for air).

The key approximation in a mathematical model for sound waves is to assume that these waves are small perturbations to the density, pressure, and velocity. We therefore write

$$\begin{aligned} p &= p_0 + \hat{p}, \\ \varrho &= \varrho_0 + \hat{\varrho}, \\ \boldsymbol{u} &= \hat{\boldsymbol{u}}, \end{aligned}$$

where we have decomposed the fields in a constant equilibrium value, corresponding to $\boldsymbol{u} = 0$, and a small perturbation marked with a hat symbol. By inserting these decompositions in (131) and (132), neglecting all product terms of small perturbations and/or their derivatives, and dropping the hat symbols, one gets the following linearized PDE system for the small perturbations in density, pressure, and velocity:

$$\varrho_t + \varrho_0 \nabla \cdot \boldsymbol{u} = 0, \tag{135}$$

$$\varrho_0 \boldsymbol{u}_t = -\nabla p \,. \tag{136}$$

Now we can eliminate ϱ_t by differentiating the relation $\varrho(p)$,

$$\varrho_t = \varrho_0 \frac{1}{\gamma} \left(\frac{p}{p_0}\right)^{1/\gamma - 1} \frac{1}{p_0} p_t = \frac{\varrho_0}{\gamma p_0} \left(\frac{p}{p_0}\right)^{1/\gamma - 1} p_t \,.$$

The product term $p^{1/\gamma-1} p_t$ can be linearized as $p_0^{1/\gamma-1} p_t$, resulting in

$$\varrho_t \approx \frac{\varrho_0}{\gamma p_0} p_t \,.$$

We then get

$$p_t + \gamma p_0 \nabla \cdot \boldsymbol{u} = 0, \tag{137}$$

$$\boldsymbol{u}_t = -\frac{1}{\varrho_0} \nabla p, \,. \tag{138}$$

Taking the divergence of (138) and differentiating (137) with respect to time gives the possibility to easily eliminate $\nabla \cdot \boldsymbol{u}_t$ and arrive at a standard, linear wave equation for p:

$$p_{tt} = c^2\nabla^2 p, \tag{139}$$

where $c = \sqrt{\gamma p_0/\varrho_0}$ is the speed of sound in the fluid.

19.6 Spherical waves

Spherically symmetric three-dimensional waves propagate in the radial direction r only so that $u = u(r,t)$. The fully three-dimensional wave equation

$$\frac{\partial^2 u}{\partial t^2} = \nabla\cdot(c^2\nabla u) + f$$

then reduces to the spherically symmetric wave equation

$$\frac{\partial^2 u}{\partial t^2} = \frac{1}{r^2}\frac{\partial}{\partial r}\left(c^2(r)r^2\frac{\partial u}{\partial t}\right) + f(r,t), \quad r\in(0,R),\ t>0\,. \tag{140}$$

One can easily show that the function $v(r,t) = ru(r,t)$ fulfills a standard wave equation in Cartesian coordinates if c is constant. To this end, insert $u = v/r$ in

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(c^2(r)r^2\frac{\partial u}{\partial t}\right)$$

to obtain

$$r\left(\frac{dc^2}{dr}\frac{\partial v}{\partial r} + c^2\frac{\partial^2 v}{\partial r^2}\right) - \frac{dc^2}{dr}v\,.$$

The two terms in the parenthesis can be combined to

$$r\frac{\partial}{\partial r}\left(c^2\frac{\partial v}{\partial r}\right),$$

which is recognized as the variable-coefficient Laplace operator in one Cartesian coordinate. The spherically symmetric wave equation in terms of $v(r,t)$ now becomes

$$\frac{\partial^2 v}{\partial t^2} = \frac{\partial}{\partial r}\left(c^2(r)\frac{\partial v}{\partial r}\right) - \frac{1}{r}\frac{dc^2}{dr}v + rf(r,t), \quad r\in(0,R),\ t>0\,. \tag{141}$$

In the case of constant wave velocity c, this equation reduces to the wave equation in a single Cartesian coordinate called r:

$$\frac{\partial^2 v}{\partial t^2} = c^2\frac{\partial^2 v}{\partial r^2} + rf(r,t), \quad r\in(0,R),\ t>0\,. \tag{142}$$

That is, any program for solving the one-dimensional wave equation in a Cartesian coordinate system can be used to solve (142), provided the source term is multiplied by the coordinate, and that we divide the Cartesian mesh solution by r to get the spherically symmetric solution. Moreover, if $r = 0$ is included in the

domain, spherical symmetry demands that $\partial u/\partial r = 0$ at $r = 0$, which means that

$$\frac{\partial u}{\partial r} = \frac{1}{r^2}\left(r\frac{\partial v}{\partial r} - v\right) = 0, \quad r = 0,$$

implying $v(0,t) = 0$ as a necessary condition. For practical applications, we exclude $r = 0$ from the domain and assume that some boundary condition is assigned at $r = \epsilon$, for some $\epsilon > 0$.

19.7 The linear shallow water equations

The next example considers water waves whose wavelengths are much lager than the depth and whose wave amplitudes are small. This class of waves may be generated by catastrophic geophysical events, such as earthquakes at the sea bottom, landslides moving into water, or underwater slides (or a combination, as earthquakes frequently release avalanches of masses). For example, a subsea earthquake will normally have an extension of many kilometers but lift the water only a few meters. The wave length will have a size dictated by the earthquake area, which is much lager than the water depth, and compared to this wave length, an amplitude of a few meters is very small. The water is essentially a thin film, and mathematically we can average the problem in the vertical direction and approximate the 3D wave phenomenon by 2D PDEs. Instead of a moving water domain in three space dimensions, we get a horizontal 2D domain with an unknown function for the surface elevation and the water depth as a variable coefficient in the PDEs.

Let $\eta(x,y,t)$ be the elevation of the water surface, $H(x,y)$ the water depth corresponding to a flat surface ($\eta = 0$), $u(x,y,t)$ and $v(x,y,t)$ the depth-averaged horizontal velocities of the water. Mass and momentum balance of the water volume give rise to the PDEs involving these quantities:

$$\eta_t = -(Hu)_x - (Hv)_x \tag{143}$$

$$u_t = -g\eta_x, \tag{144}$$

$$v_t = -g\eta_y, \tag{145}$$

where g is the acceleration of gravity. Equation (143) corresponds to mass balance while the other two are derived from momentum balance (Newton's second law).

The initial conditions associated with (143)-(145) are η, u, and v prescribed at $t = 0$. A common condition is to have some water elevation $\eta = I(x,y)$ and assume that the surface is at rest: $u = v = 0$. A subsea earthquake usually means a sufficiently rapid motion of the bottom and the water volume to say that the bottom deformation is mirrored at the water surface as an initial lift $I(x,y)$ and that $u = v = 0$.

Boundary conditions may be η prescribed for incoming, known waves, or zero normal velocity at reflecting boundaries (steep mountains, for instance):

$un_x + vn_y = 0$, where (n_x, n_y) is the outward unit normal to the boundary. More sophisticated boundary conditions are needed when waves run up at the shore, and at open boundaries where we want the waves to leave the computational domain undisturbed.

Equations (143), (144), and (145) can be transformed to a standard, linear wave equation. First, multiply (144) and (145) by H, differentiate (144)) with respect to x and (145) with respect to y. Second, differentiate (143) with respect to t and use that $(Hu)_{xt} = (Hu_t)_x$ and $(Hv)_{yt} = (Hv_t)_y$ when H is independent of t. Third, eliminate $(Hu_t)_x$ and $(Hv_t)_y$ with the aid of the other two differentiated equations. These manipulations results in a standard, linear wave equation for η:

$$\eta_{tt} = (gH\eta_x)_x + (gH\eta_y)_y = \nabla \cdot (gH\nabla\eta)\,. \tag{146}$$

In the case we have an initial non-flat water surface at rest, the initial conditions become $\eta = I(x, y)$ and $\eta_t = 0$. The latter follows from (143) if $u = v = 0$, or simply from the fact that the vertical velocity of the surface is η_t, which is zero for a surface at rest.

The system (143)-(145) can be extended to handle a time-varying bottom topography, which is relevant for modeling long waves generated by underwater slides. In such cases the water depth function H is also a function of t, due to the moving slide, and one must add a time-derivative term H_t to the left-hand side of (143). A moving bottom is best described by introducing $z = H_0$ as the still-water level, $z = B(x, y, t)$ as the time- and space-varying bottom topography, so that $H = H_0 - B(x, y, t)$. In the elimination of u and v one may assume that the dependence of H on t can be neglected in the terms $(Hu)_{xt}$ and $(Hv)_{yt}$. We then end up with a source term in (146), because of the moving (accelerating) bottom:

$$\eta_{tt} = \nabla \cdot (gH\nabla\eta) + B_{tt}\,. \tag{147}$$

The reduction of (147) to 1D, for long waves in a straight channel, or for approximately plane waves in the ocean, is trivial by assuming no change in y direction ($\partial/\partial y = 0$):

$$\eta_t = (gH\eta_x)_x + B_{tt}\,. \tag{148}$$

Wind drag on the surface. Surface waves are influenced by the drag of the wind, and if the wind velocity some meters above the surface is (U, V), the wind drag gives contributions $C_V\sqrt{U^2 + V^2}U$ and $C_V\sqrt{U^2 + V^2}V$ to (144) and (145), respectively, on the right-hand sides.

Bottom drag. The waves will experience a drag from the bottom, often roughly modeled by a term similar to the wind drag: $C_B\sqrt{u^2 + v^2}u$ on the right-hand side of (144) and $C_B\sqrt{u^2 + v^2}v$ on the right-hand side of (145). Note that in this case the PDEs (144) and (145) become nonlinear and the elimination of u and v to arrive at a 2nd-order wave equation for η is not possible anymore.

Effect of the Earth's rotation. Long geophysical waves will often be affected by the rotation of the Earth because of the Coriolis force. This force gives rise to a term fv on the right-hand side of (144) and $-fu$ on the right-hand side of (145). Also in this case one cannot eliminate u and v to work with a single equation for η. The Coriolis parameter is $f = 2\Omega \sin\phi$, where Ω is the angular velocity of the earth and ϕ is the latitude.

19.8 Waves in blood vessels

The flow of blood in our bodies is basically fluid flow in a network of pipes. Unlike rigid pipes, the walls in the blood vessels are elastic and will increase their diameter when the pressure rises. The elastic forces will then push the wall back and accelerate the fluid. This interaction between the flow of blood and the deformation of the vessel wall results in waves traveling along our blood vessels.

A model for one-dimensional waves along blood vessels can be derived from averaging the fluid flow over the cross section of the blood vessels. Let x be a coordinate along the blood vessel and assume that all cross sections are circular, though with different radius $R(x,t)$. The main quantities to compute is the cross section area $A(x,t)$, the averaged pressure $P(x,t)$, and the total volume flux $Q(x,t)$. The area of this cross section is

$$A(x,t) = 2\pi \int_0^{R(x,t)} r dr, \tag{149}$$

Let $v_x(x,t)$ be the velocity of blood averaged over the cross section at point x. The volume flux, being the total volume of blood passing a cross section per time unit, becomes

$$Q(x,t) = A(x,t)v_x(x,t) \tag{150}$$

Mass balance and Newton's second law lead to the PDEs

$$\frac{\partial A}{\partial t} + \frac{\partial Q}{\partial x} = 0, \tag{151}$$

$$\frac{\partial Q}{\partial t} + \frac{\gamma+2}{\gamma+1}\frac{\partial}{\partial x}\left(\frac{Q^2}{A}\right) + \frac{A}{\varrho}\frac{\partial P}{\partial x} = -2\pi(\gamma+2)\frac{\mu}{\varrho}\frac{Q}{A}, \tag{152}$$

where γ is a parameter related to the velocity profile, ϱ is the density of blood, and μ is the dynamic viscosity of blood.

We have three unknowns A, Q, and P, and two equations (151) and (152). A third equation is needed to relate the flow to the deformations of the wall. A common form for this equation is

$$\frac{\partial P}{\partial t} + \frac{1}{C}\frac{\partial Q}{\partial x} = 0, \tag{153}$$

where C is the compliance of the wall, given by the constitutive relation

$$C = \frac{\partial A}{\partial P} + \frac{\partial A}{\partial t}, \tag{154}$$

which require a relationship between A and P. One common model is to view the vessel wall, locally, as a thin elastic tube subject to an internal pressure. This gives the relation

$$P = P_0 + \frac{\pi h E}{(1-\nu^2)A_0}(\sqrt{A} - \sqrt{A_0}),$$

where P_0 and A_0 are corresponding reference values when the wall is not deformed, h is the thickness of the wall, and E and ν are Young's modulus and Poisson's ratio of the elastic material in the wall. The derivative becomes

$$C = \frac{\partial A}{\partial P} = \frac{2(1-\nu^2)A_0}{\pi h E}\sqrt{A_0} + 2\left(\frac{(1-\nu^2)A_0}{\pi h E}\right)^2 (P - P_0)\,. \tag{155}$$

Another (nonlinear) deformation model of the wall, which has a better fit with experiments, is

$$P = P_0 \exp\left(\beta(A/A_0 - 1)\right),$$

where β is some parameter to be estimated. This law leads to

$$C = \frac{\partial A}{\partial P} = \frac{A_0}{\beta P}\,. \tag{156}$$

Reduction to standard wave equation. It is not uncommon to neglect the viscous term on the right-hand side of (152) and also the quadratic term with Q^2 on the left-hand side. The reduced equations (152) and (153) form a first-order linear wave equation system:

$$C\frac{\partial P}{\partial t} = -\frac{\partial Q}{\partial x}, \tag{157}$$

$$\frac{\partial Q}{\partial t} = -\frac{A}{\varrho}\frac{\partial P}{\partial x}\,. \tag{158}$$

These can be combined into standard 1D wave equation PDE by differentiating the first equation with respect t and the second with respect to x,

$$\frac{\partial}{\partial t}\left(CC\frac{\partial P}{\partial t}\right) = \frac{\partial}{\partial x}\left(\frac{A}{\varrho}\frac{\partial P}{\partial x}\right),$$

which can be approximated by

$$\frac{\partial^2 Q}{\partial t^2} = c^2\frac{\partial^2 Q}{\partial x^2}, \quad c = \sqrt{\frac{A}{\varrho C}}, \tag{159}$$

where the A and C in the expression for c are taken as constant reference values.

19.9 Electromagnetic waves

Light and radio waves are governed by standard wave equations arising from Maxwell's general equations. When there are no charges and no currents, as in a vacuum, Maxwell's equations take the form

$$
\begin{aligned}
\nabla\cdot\boldsymbol{E} &= 0,\\
\nabla\cdot\boldsymbol{B} &= 0,\\
\nabla\times\boldsymbol{E} &= -\frac{\partial\boldsymbol{B}}{\partial t},\\
\nabla\times\boldsymbol{B} &= \mu_0\epsilon_0\frac{\partial\boldsymbol{E}}{\partial t},
\end{aligned}
$$

where $\epsilon_0 = 8.854187817620\cdot 10^{-12}$ (F/m) is the permittivity of free space, also known as the electric constant, and $\mu_0 = 1.2566370614\cdot 10^{-6}$ (H/m) is the permeability of free space, also known as the magnetic constant. Taking the curl of the two last equations and using the identity

$$\nabla\times(\nabla\times\boldsymbol{E}) = \nabla(\nabla\cdot\boldsymbol{E}) - \nabla^2\boldsymbol{E} = -\nabla^2\boldsymbol{E} \text{ when } \nabla\cdot\boldsymbol{E}=0,$$

immediately gives the wave equation governing the electric and magnetic field:

$$\frac{\partial^2\boldsymbol{E}}{\partial t^2} = c^2\frac{\partial^2\boldsymbol{E}}{\partial x^2}, \tag{160}$$

$$\frac{\partial^2\boldsymbol{E}}{\partial t^2} = c^2\frac{\partial^2\boldsymbol{E}}{\partial x^2}, \tag{161}$$

with $c = 1/\sqrt{\mu_0\epsilon_0}$ as the velocity of light. Each component of $\boldsymbol{E}$ and $\boldsymbol{B}$ fulfills a wave equation and can hence be solved independently.

20 Exercises

Exercise 15: Simulate waves on a non-homogeneous string

Simulate waves on a string that consists of two materials with different density. The tension in the string is constant, but the density has a jump at the middle of the string. Experiment with different sizes of the jump and produce animations that visualize the effect of the jump on the wave motion.

Hint. According to Section 19.1, the density enters the mathematical model as ϱ in $\varrho u_{tt} = Tu_{xx}$, where T is the string tension. Modify, e.g., the `wave1D_u0v.py` code to incorporate the tension and two density values. Make a mesh function `rho` with density values at each spatial mesh point. A value for the tension may be 150 N. Corresponding density values can be computed from the wave velocity estimations in the `guitar` function in the `wave1D_u0v.py` file.
Filename: `wave1D_u0_sv_discont.py`.

Exercise 16: Simulate damped waves on a string

Formulate a mathematical model for damped waves on a string. Use data from Section 3.4, and tune the damping parameter so that the string is very close to the rest state after 15 s. Make a movie of the wave motion. Filename: `wave1D_u0_sv_damping.py`.

Exercise 17: Simulate elastic waves in a rod

A hammer hits the end of an elastic rod. The exercise is to simulate the resulting wave motion using the model (121) from Section 19.3. Let the rod have length L and let the boundary $x = L$ be stress free so that $\sigma_{xx} = 0$, implying that $\partial u/\partial x = 0$. The left end $x = 0$ is subject to a strong stress pulse (the hammer), modeled as

$$\sigma_{xx}(t) = \begin{cases} S, & 0 < t \leq t_s, \\ 0, & t > t_s \end{cases}$$

The corresponding condition on u becomes $u_x = S/E$ for $t \leq t_s$ and zero afterwards (recall that $\sigma_{xx} = Eu_x$). This is a non-homogeneous Neumann condition, and you will need to approximate this condition and combine it with the scheme (the ideas and manipulations follow closely the handling of a non-zero initial condition $u_t = V$ in wave PDEs or the corresponding second-order ODEs for vibrations). Filename: `wave_rod.py`.

Exercise 18: Simulate spherical waves

Implement a model for spherically symmetric waves using the method described in Section 19.6. The boundary condition at $r = 0$ must be $\partial u/\partial r = 0$, while the condition at $r = R$ can either be $u = 0$ or a radiation condition as described in Problem 21. The $u = 0$ condition is sufficient if R is so large that the amplitude of the spherical wave has become insignificant. Make movie(s) of the case where the source term is located around $r = 0$ and sends out pulses

$$f(r,t) = \begin{cases} Q\exp\left(-\frac{r^2}{2\Delta r^2}\right)\sin\omega t, & \sin\omega t \geq 0 \\ 0, & \sin\omega t < 0 \end{cases}$$

Here, Q and ω are constants to be chosen.

Hint. Use the program `wave1D_u0v.py` as a starting point. Let `solver` compute the v function and then set $u = v/r$. However, $u = v/r$ for $r = 0$ requires special treatment. One possibility is to compute `u[1:]  = v[1:]/r[1:]` and then set `u[0]=u[1]`. The latter makes it evident that $\partial u/\partial r = 0$ in a plot. Filename: `wave1D_spherical.py`.

Exercise 19: Explain why numerical noise occurs

The experiments performed in Exercise 8 shows considerable numerical noise in the form of non-physical waves, especially for $s_f = 4$ and the plug pulse or the half a "cosinehat" pulse. The noise is much less visible for a Gaussian pulse. Run the case with the plug and half a "cosinehat" pulses for $s_f = 1$, $C = 0.9, 0.25$, and $N_x = 40, 80, 160$. Use the numerical dispersion relation to explain the observations. Filename: `pulse1D_analysis.pdf`.

Exercise 20: Investigate harmonic averaging in a 1D model

Harmonic means are often used if the wave velocity is non-smooth or discontinuous. Will harmonic averaging of the wave velocity give less numerical noise for the case $s_f = 4$ in Exercise 8? Filenames: `pulse1D_harmonic.pdf`, `pulse1D_harmonic.py`.

Problem 21: Implement open boundary conditions

To enable a wave to leave the computational domain and travel undisturbed through the boundary $x = L$, one can in a one-dimensional problem impose the following condition, called a *radiation condition* or *open boundary condition*:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0\,. \tag{162}$$

The parameter c is the wave velocity.

Show that (162) accepts a solution $u = g_R(x - ct)$ (right-going wave), but not $u = g_L(x + ct)$ (left-going wave). This means that (162) will allow any right-going wave $g_R(x - ct)$ to pass through the boundary undisturbed.

A corresponding open boundary condition for a left-going wave through $x = 0$ is

$$\frac{\partial u}{\partial t} - c\frac{\partial u}{\partial x} = 0\,. \tag{163}$$

a) A natural idea for discretizing the condition (162) at the spatial end point $i = N_x$ is to apply centered differences in time and space:

$$[D_{2t}u + cD_{2x}u = 0]^n_i, \quad i = N_x\,. \tag{164}$$

Eliminate the fictitious value $u^n_{N_x+1}$ by using the discrete equation at the same point.

The equation for the first step, u^1_i, is in principle also affected, but we can then use the condition $u_{N_x} = 0$ since the wave has not yet reached the right boundary.

b) A much more convenient implementation of the open boundary condition at $x = L$ can be based on an explicit discretization

$$[D_t^+ u + cD_x^- u = 0]_i^n, \quad i = N_x\,. \tag{165}$$

From this equation, one can solve for $u_{N_x}^{n+1}$ and apply the formula as a Dirichlet condition at the boundary point. However, the finite difference approximations involved are of first order.

Implement this scheme for a wave equation $u_{tt} = c^2 u_{xx}$ in a domain $[0, L]$, where you have $u_x = 0$ at $x = 0$, the condition (162) at $x = L$, and an initial disturbance in the middle of the domain, e.g., a plug profile like

$$u(x, 0) = \begin{cases} 1, & L/2 - \ell \leq x \leq L/2 + \ell, \\ 0, \text{otherwise} \end{cases}$$

Observe that the initial wave is split in two, the left-going wave is reflected at $x = 0$, and both waves travel out of $x = L$, leaving the solution as $u = 0$ in $[0, L]$. Use a unit Courant number such that the numerical solution is exact. Make a movie to illustrate what happens.

Because this simplified implementation of the open boundary condition works, there is no need to pursue the more complicated discretization in a).

Hint. Modify the solver function in `wave1D_dn.py`.

c) Add the possibility to have either $u_x = 0$ or an open boundary condition at the left boundary. The latter condition is discretized as

$$[D_t^+ u - cD_x^+ u = 0]_i^n, \quad i = 0, \tag{166}$$

leading to an explicit update of the boundary value u_0^{n+1}.

The implementation can be tested with a Gaussian function as initial condition:

$$g(x; m, s) = \frac{1}{\sqrt{2\pi}s} e^{-\frac{(x-m)^2}{2s^2}}\,.$$

Run two tests:

1. Disturbance in the middle of the domain, $I(x) = g(x; L/2, s)$, and open boundary condition at the left end.
2. Disturbance at the left end, $I(x) = g(x; 0, s)$, and $u_x = 0$ as symmetry boundary condition at this end.

Make nose tests for both cases, testing that the solution is zero after the waves have left the domain.

d) In 2D and 3D it is difficult to compute the correct wave velocity normal to the boundary, which is needed in generalizations of the open boundary conditions in higher dimensions. Test the effect of having a slightly wrong wave velocity in (165). Make a movies to illustrate what happens.
Filename: `wave1D_open_BC.py`.

Remarks. The condition (162) works perfectly in 1D when c is known. In 2D and 3D, however, the condition reads $u_t + c_x u_x + c_y u_y = 0$, where c_x and c_y are the wave speeds in the x and y directions. Estimating these components (i.e., the direction of the wave) is often challenging. Other methods are normally used in 2D and 3D to let waves move out of a computational domain.

Exercise 22: Implement periodic boundary conditions

It is frequently of interest to follow wave motion over large distances and long times. A straightforward approach is to work with a very large domain, but might lead to a lot of computations in areas of the domain where the waves cannot be noticed. A more efficient approach is to let a right-going wave out of the domain and at the same time let it enter the domain on the left. This is called a *periodic boundary condition*.

The boundary condition at the right end $x = L$ is an open boundary condition (see Exercise 21) to let a right-going wave out of the domain. At the left end, $x = 0$, we apply, in the beginning of the simulation, either a symmetry boundary condition (see Exercise 7) $u_x = 0$, or an open boundary condition.

This initial wave will split in two and either reflected or transported out of the domain at $x = 0$. The purpose of the exercise is to follow the right-going wave. We can do that with a *periodic boundary condition*. This means that when the right-going wave hits the boundary $x = L$, the open boundary condition lets the wave out of the domain, but at the same time we use a boundary condition on the left end $x = 0$ that feeds the outgoing wave into the domain again. This periodic condition is simply $u(0) = u(L)$. The switch from $u_x = 0$ or an open boundary condition at the left end to a periodic condition can happen when $u(L, t) > \epsilon$, where $\epsilon = 10^{-4}$ might be an appropriate value for determining when the right-going wave hits the boundary $x = L$.

The open boundary conditions can conveniently be discretized as explained in Exercise 21. Implement the described type of boundary conditions and test them on two different initial shapes: a plug $u(x, 0) = 1$ for $x \leq 0.1$, $u(x, 0) = 0$ for $x > 0.1$, and a Gaussian function in the middle of the domain: $u(x, 0) = \exp\left(-\frac{1}{2}(x - 0.5)^2/0.05\right)$. The domain is the unit interval $[0, 1]$. Run these two shapes for Courant numbers 1 and 0.5. Assume constant wave velocity. Make movies of the four cases. Reason why the solutions are correct. Filename: `periodic.py`.

Problem 23: Earthquake-generated tsunami over a subsea hill

A subsea earthquake leads to an immediate lift of the water surface, see Figure 10. The lifted water surface splits into two tsunamis, one traveling to the right and one to the left, as depicted in Figure 11. Since tsunamis are normally very long waves, compared to the depth, with a small amplitude, compared to the wave length, the wave equation model described in Section 19.7 is relevant:

$$\eta_{tt} = (gH(x)\eta_x)_x,$$

where g is the acceleration of gravity, and $H(x)$ is the still water depth.

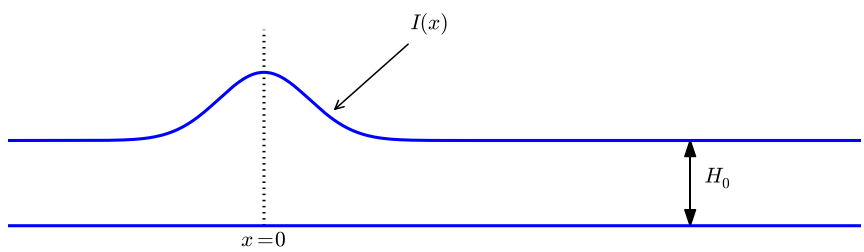

Figure 10: Sketch of initial water surface due to a subsea earthquake.

To simulate the right-going tsunami, we can impose a symmetry boundary at $x = 0$: $\partial\eta\ \partial x = 0$. We then simulate the wave motion in $[0, L]$. Unless the ocean ends at $x = L$, the waves should travel undisturbed through the boundary $x = L$. A radiation condition as explained in Problem 21 can be used for this purpose. Alternatively, one can just stop the simulations before the wave hits the boundary at $x = L$. In that case it does not matter what kind of boundary condition we use at $x = L$. Imposing $\eta = 0$ and stopping the simulations when $|\eta_i^n| > \epsilon$, $i = N_x - 1$, is a possibility (ϵ is a small parameter).

The shape of the initial surface can be taken as a Gaussian function,

$$I(x; I_0, I_a, I_m, I_s) = I_0 + I_a \exp\left(-\left(\frac{x - I_m}{I_s}\right)^2\right), \tag{167}$$

with $I_m = 0$ reflecting the location of the peak of $I(x)$ and I_s being a measure of the width of the function $I(x)$ (I_s is $\sqrt{2}$ times the standard deviation of the familiar normal distribution curve).

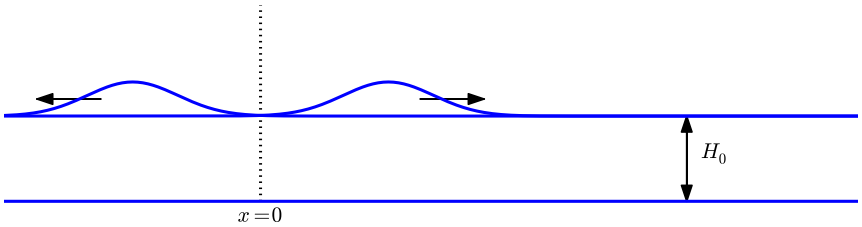

Figure 11: An initial surface elevation is split into two waves.

Now we extend the problem with a hill at the sea bottom, see Figure 12. The wave speed $c = \sqrt{gH(x)} = \sqrt{g(H_0 - B(x))}$ will then be reduced in the shallow water above the hill.

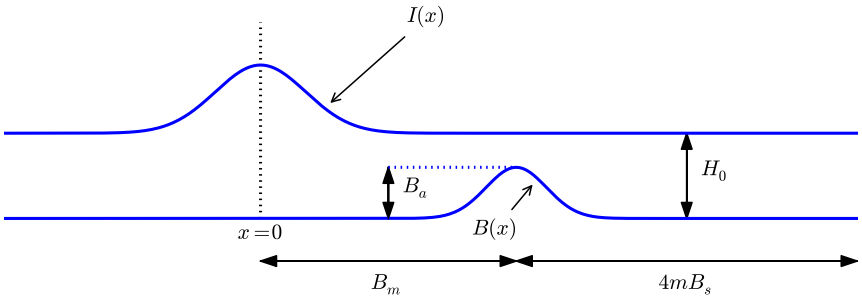

Figure 12: Sketch of an earthquake-generated tsunami passing over a subsea hill.

One possible form of the hill is a Gaussian function,

$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a \exp\left(-\left(\frac{x - B_m}{B_s}\right)^2\right), \tag{168}$$

but many other shapes are also possible, e.g., a "cosine hat" where

$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a \cos\left(\pi \frac{x - B_m}{2B_s}\right), \tag{169}$$

when $x \in [B_m - B_s, B_m + B_s]$ while $B = B_0$ outside this interval.

Also an abrupt construction may be tried:

$$B(x; B_0, B_a, B_m, B_s) = B_0 + B_a, \tag{170}$$

for $x \in [B_m - B_s, B_m + B_s]$ while $B = B_0$ outside this interval.

The `wave1D_dn_vc.py` program can be used as starting point for the implementation. Visualize both the bottom topography and the water surface elevation in the same plot. Allow for a flexible choice of bottom shape: (168), (169), (170), or $B(x) = B_0$ (flat).

The purpose of this problem is to explore the quality of the numerical solution η_i^n for different shapes of the bottom obstruction. The "cosine hat" and the box-shaped hills have abrupt changes in the derivative of $H(x)$ and are more likely to generate numerical noise than the smooth Gaussian shape of the hill. Investigate if this is true. Filenames: `tsunami1D_hill.py`, `tsunami1D_hill.pdf`.

Problem 24: Earthquake-generated tsunami over a 3D hill

This problem extends Problem 23 to a three-dimensional wave phenomenon, governed by the 2D PDE (146). We assume that the earthquake arise from a fault along the line $x = 0$ in the xy-plane so that the initial lift of the surface can be taken as $I(x)$ in Problem 23. That is, a plane wave is propagating to the right, but will experience bending because of the bottom.

The bottom shape is now a function of x and y. An "elliptic" Gaussian function in two dimensions, with its peak at (B_{mx}, B_{my}), generalizes (168):

$$B(x; B_0, B_a, B_{mx}, B_{my}, B_s, b) = B_0 + B_a \exp\left(-\left(\frac{x - B_{mx}}{B_s}\right)^2 - \left(\frac{y - B_{my}}{bB_s}\right)^2\right), \tag{171}$$

where b is a scaling parameter: $b = 1$ gives a circular Gaussian function with circular contour lines, while $b \neq 1$ gives an elliptic shape with elliptic contour lines.

The "cosine hat" (169) can also be generalized to

$$B(x; B_0, B_a, B_{mx}, B_{my}, B_s) = B_0 + B_a \cos\left(\pi \frac{x - B_{mx}}{2B_s}\right) \cos\left(\pi \frac{y - B_{my}}{2B_s}\right), \tag{172}$$

when $0 \leq \sqrt{x^2 + y^2} \leq B_s$ and $B = B_0$ outside this circle.

A box-shaped obstacle means that

$$B(x; B_0, B_a, B_m, B_s, b) = B_0 + B_a \tag{173}$$

for x and y inside a rectangle

$$B_{mx} - B_s \leq x \leq B_{mx} + B_s, \quad B_{my} - bB_s \leq y \leq B_{my} + bB_s,$$

and $B = B_0$ outside this rectangle. The b parameter controls the rectangular shape of the cross section of the box.

Note that the initial condition and the listed bottom shapes are symmetric around the line $y = B_{my}$. We therefore expect the surface elevation also to be symmetric with respect to this line. This means that we can halve the computational domain by working with $[0, L_x] \times [0, B_{my}]$. Along the upper boundary, $y = B_{my}$, we must impose the symmetry condition $\partial\eta/\partial n = 0$. Such a symmetry condition ($-\eta_x = 0$) is also needed at the $x = 0$ boundary because the initial condition has a symmetry here. At the lower boundary $y = 0$ we also set a Neumann condition (which becomes $-\eta_y = 0$). The wave motion is to be simulated until the wave hits the reflecting boundaries where $\partial\eta/\partial n = \eta_x = 0$ (one can also set $\eta = 0$ - the particular condition does not matter as long as the simulation is stopped before the wave is influenced by the boundary condition).

Visualize the surface elevation. Investigate how different hill shapes, different sizes of the water gap above the hill, and different resolutions $\Delta x = \Delta y = h$ and Δt influence the numerical quality of the solution. Filenames: `tsunami2D_hill.py`, `tsunami2D_hill.pdf`.

Problem 25: Investigate Matplotlib for visualization

Play with native Matplotlib code for visualizing 2D solutions of the wave equation with variable wave velocity. See if there are effective ways to visualize both the solution and the wave velocity. Filename: `tsunami2D_hill_mpl.py`.

Problem 26: Investigate visualization packages

Create some fancy 3D visualization of the water waves *and* the subsea hill in Problem 24. Try to make the hill transparent. Possible visualization tools are

- Mayavi
- Paraview
- OpenDX

Filename: `tsunami2D_hill_viz.py`.

Problem 27: Implement loops in compiled languages

Extend the program from Problem 24 such that the loops over mesh points, inside the time loop, are implemented in compiled languages. Consider implementations in Cython, Fortran via `f2py`, C via Cython, C via `f2py`, C/C++ via Instant, and C/C++ via `scipy.weave`. Perform efficiency experiments to investigate the relative performance of the various implementations. It is often advantageous

to normalize CPU times by the fastest method on a given mesh. Filename: `tsunami2D_hill_compiled.py`.

Exercise 28: Simulate seismic waves in 2D

The goal of this exercise is to simulate seismic waves using the PDE model (130) in a 2D xz domain with geological layers. Introduce m horizontal layers of thickness h_i, $i = 0, \ldots, m-1$. Inside layer number i we have a vertical wave velocity $c_{z,i}$ and a horizontal wave velocity $c_{h,i}$. Make a program for simulating such 2D waves. Test it on a case with 3 layers where

$$c_{z,0} = c_{z,1} = c_{z,2}, \quad c_{h,0} = c_{h,2}, \quad c_{h,1} \ll c_{h,0}\,.$$

Let s be a localized point source at the middle of the Earth's surface (the upper boundary) and investigate how the resulting wave travels through the medium. The source can be a localized Gaussian peak that oscillates in time for some time interval. Place the boundaries far enough from the expanding wave so that the boundary conditions do not disturb the wave. Then the type of boundary condition does not matter, except that we physically need to have $p = p_0$, where p_0 is the atmospheric pressure, at the upper boundary. Filename: `seismic2D.py`.

Project 29: Model 3D acoustic waves in a room

The equation for sound waves in air is derived in Section 19.5 and reads

$$p_{tt} = c^2\nabla^2 p,$$

where $p(x, y, z, t)$ is the pressure and c is the speed of sound, taken as 340 m/s. However, sound is absorbed in the air due to relaxation of molecules in the gas. A model for simple relaxation, valid for gases consisting only of one type of molecules, is a term $c^2\tau_s\nabla^2 p_t$ in the PDE, where τ_s is the relaxation time. If we generate sound from, e.g., a loudspeaker in the room, this sound source must also be added to the governing equation.

The PDE with the mentioned type of damping and source then becomes

$$p_t t = c^2\nabla^p + c^2\tau_s\nabla^2 p_t + f, \tag{174}$$

where $f(x, y, z, t)$ is the source term.

The walls can absorb some sound. A possible model is to have a "wall layer" (thicker than the physical wall) outside the room where c is changed such that some of the wave energy is reflected and some is absorbed in the wall. The absorption of energy can be taken care of by adding a damping term bp_t in the equation:

$$p_t t + bp_t = c^2\nabla^p + c^2\tau_s\nabla^2 p_t + f\,. \tag{175}$$

Typically, $b = 0$ in the room and $b > 0$ in the wall. A discontinuity in b or c will give rise to reflections. It can be wise to use a constant c in the wall to control reflections because of the discontinuity between c in the air and in the wall, while b is gradually increased as we go into the wall to avoid reflections because of rapid changes in b. At the outer boundary of the wall the condition $p = 0$ or $\partial p/\partial n = 0$ can be imposed. The waves should anyway be approximately dampened to $p = 0$ this far out in the wall layer.

There are two strategies for discretizing the $\nabla^2 p_t$ term: using a center difference between times $n+1$ and $n-1$ (if the equation is sampled at level n), or use a one-sided difference based on levels n and $n-1$. The latter has the advantage of not leading to any equation system, while the former is second-order accurate as the scheme for the simple wave equation $p_t t = c^2\nabla^2 p$. To avoid an equation system, go for the one-sided difference such that the overall scheme becomes explicit and only of first order in time.

Develop a 3D solver for the specified PDE and introduce a wall layer. Test the solver with the method of manufactured solutions. Make some demonstrations where the wall reflects and absorbs the waves (reflection because of discontinuity in b and absorption because of growing b). Experiment with the impact of the τ_s parameter. Filename: `acoustics.py`.

Project 30: Solve a 1D transport equation

We shall study the wave equation

$$u_t + cu_x = 0, \quad x \in (0, L],\ t \in (0, T], \tag{176}$$

with initial condition

$$u(x, 0) = I(x), \quad x \in [0, L], \tag{177}$$

and *one* periodic boundary condition

$$u(0, t) = u(L, t)\,. \tag{178}$$

This boundary condition means that what goes out of the domain at $x = L$ comes in at $x = 0$. Roughly speaking, we need only one boundary condition because of the spatial derivative is of first order only.

Physical interpretation. The parameter c can be constant or variable, $c = c(x)$. The equation (176) arises in *transport* problems where a quantity u, which could be temperature or concentration of some contaminant, is transported with the velocity c of a fluid. In addition to the transport imposed by "travelling with the fluid", u may also be transported by diffusion (such as heat conduction or Fickian diffusion), but we have in the model $u_t + cu_x$ assumed that diffusion effects are negligible, which they often are.

A widely used numerical scheme for (176) applies a forward difference in time and a backward difference in space when $c > 0$:

$$[D_t^+ u + cD_x^- u = 0]_i^n \,. \tag{179}$$

For $c < 0$ we use a forward difference in space: $[cD_x^+ u]_i^n$.

We shall hereafter assume that $= c(x) > 0$.

To compute (184) we need to integrate $1/c$ to obtain C and then compute the inverse of C.

The inverse function computation can be easily done if we first think discretely. Say we have some function $y = g(x)$ and seeks its inverse. Plotting (x_i, y_i), where $y_i = g(x_i)$ for some mesh points x_i, displays g as a function of x. The inverse function is simply x as a function of g, i.e., the curve with points (y_i, x_i). We can therefore quickly compute points at the curve of the inverse function. One way of extending these points to a continuous function is to assume a linear variation (known as linear interpolation) between the points (which actually means to draw straight lines between the points, exactly as done by a plotting program).

The function `wrap2callable` in `scitools.std` can take a set of points and return a continuous function that corresponds to linear variation between the points. The computation of the inverse of a function g on $[0, L]$ can then be done by

```
def inverse(g, domain, resolution=101):
    x = linspace(domain[0], domain[L], resolution)
    y = g(x)
    from scitools.std import wrap2callable
    g_inverse = wrap2callable((y, x))
    return g_inverse
```

To compute $C(x)$ we need to integrate $1/c$, which can be done by a Trapezoidal rule. Suppose we have computed $C(x_i)$ and need to compute $C(x_{i+1})$. Using the Trapezoidal rule with m subintervals over the integration domain $[x_i, x_{i+1}]$ gives

$$C(x_{i+1}) = C(x_i) + \int_{x_i}^{x_{i+1}} \frac{dx}{c} \approx h\left(\frac{1}{2}\frac{1}{c(x_i)} + \frac{1}{2}\frac{1}{c(x_{i+1})} + \sum_{j=1}^{m-1}\frac{1}{c(x_i + jh)}\right), \tag{180}$$

where $h = (x_{i+1} - x_i)/m$ is the length of the subintervals used for the integral over $[x_i, x_{i+1}]$. We observe that (180) is a *difference equation* which we can solve by repeatedly applying (180) for $i = 0, 1, \ldots, N_x - 1$ if a mesh $x_0, x_, \ldots, x_{N_x}$ is prescribed. Note that $C(0) = 0$.

a) Show that under the assumption of $a = \text{const}$,

$$u(x, t) = I(x - ct) \tag{181}$$

fulfills the PDE as well as the initial and boundary condition (provided $I(0) = I(L)$).

b) Set up a computational algorithm and implement it in a function. Assume a is constant and positive.

c) Test implementation by using the remarkable property that the numerical solution is exact at the mesh points if $\Delta t = c^{-1}\Delta x$.

d) Make a movie comparing the numerical and exact solution for the following two choices of initial conditions:

$$I(x) = \left[\sin\left(\pi\frac{x}{L}\right)\right]^{2n} \tag{182}$$

where n is an integer, typically $n = 5$, and

$$I(x) = \exp\left(-\frac{(x - L/2)^2}{2\sigma 2}\right). \tag{183}$$

Choose $\Delta t = c^{-1}\Delta x, 0.9c^{-1}\Delta x, 0.5c^{-1}\Delta x$.

e) The performance of the suggested numerical scheme can be investigated by analyzing the numerical dispersion relation. Analytically, we have that the *Fourier component*

$$u(x,t) = e^{i(kx-\omega t)},$$

is a solution of the PDE if $\omega = kc$. This is the *analytical dispersion relation.* A complete solution of the PDE can be built by adding up such Fourier components with different amplitudes, where the initial condition I determines the amplitudes. The solution u is then represented by a Fourier series.

A similar discrete Fourier component at (x_p, t_n) is

$$u_p^q = e^{i(kp\Delta x - \tilde{\omega} n\Delta t)},$$

where in general $\tilde{\omega}$ is a function of k, Δt, and Δx, and differs from the exact $\omega = kc$.

Insert the discrete Fourier component in the numerical scheme and derive an expression for $\tilde{\omega}$, i.e., the discrete dispersion relation. Show in particular that if the $\Delta t/(c\Delta x) = 1$, the discrete solution coincides with the exact solution at the mesh points, regardless of the mesh resolution (!). Show that if the stability condition

$$\frac{\Delta t}{c\Delta x} \leq 1,$$

the discrete Fourier component cannot grow (i.e., $\tilde{\omega}$ is real).

f) Write a test for your implementation where you try to use information from the numerical dispersion relation.

g) Set up a computational algorithm for the variable coefficient case and implement it in a function. Make a test that the function works for constant a.

h) It can be shown that for an observer moving with velocity $c(x)$, u is constant. This can be used to derive an exact solution when a varies with x. Show first that

$$u(x,t) = f(C(x) - t), \tag{184}$$

where

$$C'(x) = \frac{1}{c(x)},$$

is a solution of (176) for any differentiable function f.

i) Use the initial condition to show that an exact solution is

$$u(x,t) = I(C^{-1}(C(x) - t)),$$

with C^{-1} being the inverse function of $C = \int c^1 dx$. Since $C(x)$ is an integral $\int_0^x (1/c)dx$, $C(x)$ is monotonically increasing and there exists hence an inverse function C^{-1} with values in $[0, L]$.

j) Implement a function for computing $C(x_i)$ and one for computing $C^{-1}(x)$ for any x. Use these two functions for computing the exact solution $I(C^{-1}(C(x)-t))$. End up with a function `u_exact_variable_c(x, n, c, I)` that returns the value of $I(C^{-1}(C(x) - t_n))$.

k) Make movies showing a comparison of the numerical and exact solutions for the two initial conditions (182) and (30). Choose $\Delta t = \Delta x / \max_{0,L} c(x)$ and the velocity of the medium as

1. $c(x) = 1 + \epsilon \sin(k\pi x/L)$, $\epsilon < 1$,
2. $c(x) = 1 + I(x)$, where I is given by (182) or (30).

The PDE $u_t + cu_x = 0$ expresses that the initial condition $I(x)$ is transported with velocity $c(x)$.
Filename: `advec1D.py`.

Problem 31: General analytical solution of a 1D damped wave equation

We consider an initial-boundary value problem for the damped wave equation:

$$\begin{aligned}
u_{tt} + bu_t &= c^2 u_{xx}, && x \in (0,L),\ t \in (0,T] \\
u(0,t) &= 0, \\
u(L,t) &= 0, \\
u(x,0) &= I(x), \\
u_t(x,0) &= V(x)\,.
\end{aligned}$$

Here, $b \geq 0$ and c are given constants. The aim is to derive a general analytical solution of this problem. Familiarity with the method of separation of variables for solving PDEs will be assumed.

a) Seek a solution on the form $u(x,t) = X(x)T(t)$. Insert this solution in the PDE and show that it leads to two differential equations for X and T:

$$T'' + bT' + \lambda T = 0, \quad c^2 X'' + \lambda X = 0,$$

with $X(0) = X(L) = 0$ as boundary conditions, and λ as a constant to be determined.

b) Show that $X(x)$ is on the form

$$X_n(x) = C_n \sin kx, \quad k = \frac{n\pi}{L}, \quad n = 1, 2, \ldots$$

where C_n is an arbitrary constant.

c) Under the assumption that $(b/2)^2 < k^2$, show that $T(t)$ is on the form

$$T_n(t) = e^{-\frac{1}{2}bt}(a_n \cos\omega t + b_n \sin\omega t), \quad \omega = \sqrt{k^2 - \frac{1}{4}b^2}, \quad n = 1, 2, \ldots$$

The complete solution is then

$$u(x,t) = \sum_{n=1}^{\infty} \sin kx e^{-\frac{1}{2}bt}(A_n \cos\omega t + B_n \sin\omega t),$$

where the constants A_n and B_n must be computed from the initial conditions.

d) Derive a formula for A_n from $u(x,0) = I(x)$ and developing $I(x)$ as a sine Fourier series on $[0, L]$.

e) Derive a formula for B_n from $u_t(x,0) = V(x)$ and developing $V(x)$ as a sine Fourier series on $[0, L]$.

f) Calculate A_n and B_n from vibrations of a string where $V(x) = 0$ and

$$I(x) = \begin{cases} ax/x_0, & x < x_0, \\ a(L-x)/(L-x_0), & \text{otherwise} \end{cases} \tag{185}$$

g) Implement the series for $u(x,t)$ in a function `u_series(x, t, tol=1E-10)`, where `tol` is a tolerance for truncating the series. Simply sum the terms until $|a_n|$ and $|b_b|$ both are less than `tol`.

h) What will change in the derivation of the analytical solution if we have $u_x(0,t) = u_x(L,t) = 0$ as boundary conditions? And how will you solve the problem with $u(0,t) = 0$ and $u_x(L,t) = 0$?
Filename: `damped_wave1D.pdf`.

Problem 32: General analytical solution of a 2D damped wave equation

Carry out Problem 31 in the 2D case: $u_{tt} + bu_t = c^2(u_{xx} + u_{yy})$, where $(x, y) \in (0, L_x) \times (0, L_y)$. Assume a solution on the form $u(x, y, t) = X(x)Y(y)T(t)$. Filename: `damped_wave2D.pdf`.

Index

arithmetic mean, 39
array slices, 22
averaging
    arithmetic, 39
    geometric, 39
    harmonic, 39

boundary condition
    open (radiation), 103
boundary conditions
    Dirichlet, 30
    Neumann, 30
    periodic, 105

C extension module, 76
C/Python array storage, 81
column-major ordering, 81
Courant number, 56
Cython, 73
`cython -a` (Python-C translation in HTML), 75

declaration of variables in Cython, 74
Dirichlet conditions, 30
discrete Fourier transform, 53
`distutils`, 76

Fortran array storage, 81
Fortran subroutine, 78
Fourier series, 53
Fourier transform, 53

geometric mean, 39

harmonic average, 39
homogeneous Dirichlet conditions, 30
homogeneous Neumann conditions, 30

index set notation, 33, 69

lambda function (Python), 25

mesh
    finite differences, 5
mesh function, 6

Neumann conditions, 30
`nose` tests, 17

open boundary condition, 103

periodic boundary conditions, 105

radiation condition, 103
row-major ordering, 81

scalar code, 22
`setup.py`, 76
slice, 22
software testing
    nose, 17
stability criterion, 56
stencil
    1D wave equation, 6
    Neumann boundary, 31

unit testing, 17

vectorization, 22

wave equation
    1D, 5
    1D, analytical properties, 51
    1D, exact numerical solution, 54
    1D, finite difference method, 5
    1D, implementation, 15
    1D, stability, 56
    2D, implementation, 67
waves
    on a string, 5
wrapper code, 78